Introduction

Purpose

This guide describes the Telephony Service Provider Interface (TSPI) for Telephony and explains how to create service providers for the Microsoft Windows family of operating systems that support the Microsoft Win32 application programming interface (API). The Windows TSPI defines the external interface of service providers that are implemented by vendors of telephony equipment. In Windows Telephony, service providers provide access to vendor-specific equipment through a procedure-oriented DLL interface. Installing a service provider allows Win32-based applications that use elements of telephony to access the corresponding telephony equipment.

It is assumed that you have read the *Windows Telephony Application Programmer's Reference*. This book describes many of the basic concepts that you need to understand, such as the telephony model and the telphony application programming interface (TAPI), to successful develop a telephony service provider.

Audience

This document is intended to be used by developers experienced with programming telephonic applications, and with programming applications for the Microsoft Windows environment.

Document Overview

This guide presents information on how to use Telephony SPI (TSPI) functions, messages, and data types. Following this introduction are the following chapters:

- Chapter 1, *Overview of the Telephony SPI*, provides an overview of the Telephony SPI. It describes basic concepts, the usage model underlying the Telephony SPI, its relation to other Windows SPIs, and an overview of the functions, messages, and data structures provided by the Telephony SPI.
- Chapter 2, *The Telephony Programming Model*, explains the concepts programmers should know before they begin developing a telephonic service provider. It includes information such as a description of the relationships between telephonic applications, TAPI.DLL, and service providers.
- Chapter 3, *Programming Service Providers: Getting Started*, outlines the various elements of the Telephony SDK.
- Chapter 4, *Concepts of Line Devices*, explains the use and functions pertaining to one of the two device classes defined by TAPI: the line device.
- Chapter 5, *Concepts of Phone Devices*, explains the use and functions pertaining to one of the two device classes defined by TAPI: the phone device.

Chapter 5 is followed by several chapters that constitute the TSPI reference section. Each of these chapters provides a detailed listing of the functions, messages, or data types used by the Telephony SPI, either exclusively or in conjunction with the Telephony API. These chapters, which also comprise the online TSPI reference (a Windows Help file), are the following:

- Chapter 6, *Service-Provider Functions*
- Chapter 7, *TSPI Line-Related Functions*
- Chapter 8, *TSPI Line-Related Messages*
- Chapter 9, *TSPI Data Types*
- Chapter 10, *TSPI Line-Device Data Structures*
- Chapter 11, *TSPI Line-Device Constants*
- Chapter 12, *TSPI Phone-Related Functions*
- Chapter 13, *TSPI Phone-Related Messages*
- Chapter 14, *TSPI Phone-Related Data Structures*
- Chapter 15, *TSPI Phone-Related Constants*

Following the above reference chapters are three appendices containing supplementary reference information for using TSPI:

- Appendix A, *Configuration Information and TELEPHON.INI*, describes the purpose of the TELEPHON.INI file and why telephonic applications would need to communicate with it.
- Appendix B, *Call States and Events*, describes the call states through which a call transitions as it is established or disconnected.
- Appendix C, *Glossary of Telephony Terminology*, defines terms useful to developers of telephony applications.

Document Conventions

The document conventions used in this document are as follows:

Bold text	Bold letters indicate terms that you must enter exactly as shown. These include function, structure, and structure member names.
ALL CAPS	All capital letters indicates names that you must enter exactly as show. These include message and constant names.
Italic text	In introductory and explanatory text, italicized words indicate that a key term or concept is being introduced. In describing functions and messages, words in italics indicate a place holder; you are expected to provide the actual value.
Monospaced text	Monospaced typeface denotes code samples and syntax.

TSPI Update for Windows 95

TSPI Update for Windows 95

Introduction

This chapter describes the additions and changes to the Telephony service provider interface (SPI) for Windows 95. TSPI includes new functions and messages as well as some changes to existing structures and constants to correspond with the changes in TAPI.

Reference

Functions

- TSPI_providerEnumDevices
- TSPI_providerCreateLineDevice
- TSPI_providerCreatePhoneDevice
- TSPI_lineDropOnClose
- TSPI_lineDropNoOwner
- TSPI_lineSetCurrentLocation
- TSPI_lineConfigDialogEdit
- TSPI_lineReleaseUserUserInfo

TSPI_providerEnumDevices

[New - Windows 95]

```
LONG TSPIAPI TSPI_providerEnumDevices(DWORD dwPermanentProviderID,
    LPDWORD lpdwNumLines, LPDWORD lpdwNumPhones, HPROVIDER hProvider,
    LINEEVENT lpfnLineCreateProc, PHONEEVENT lpfnPhoneCreateProc)
```

TAPI.DLL calls this function before TSPI_providerInit to determine the number of line and phone devices supported by the service provider.

- Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

  LINEERR_NOMEM

  LINEERR_OPERATIONFAILED

This function must be declared in the DEF file of the service provider with the ordinal value 595.

dwPermanentProviderID

Specifies the permanent ID, unique within the service providers on this system, of the service provider being initialized.

lpdwNumLines

Specifies a far pointer to a DWORD-sized memory location into which the service provider must write the number of line devices it is configured to support. TAPI.DLL initializes the value to zero, so if the service provider fails to write a different value, the value 0 is assumed.

lpdwNumPhones

Specifies a far pointer to a DWORD-sized memory location into which the service provider must write the number of phone devices it is configured to support. TAPI.DLL initializes the value to zero, so if the service provider fails to write a different value, the value 0 is assumed.

hProvider

Specifies an opaque DWORD-sized value which uniquely identifies this instance of this service provider during this execution of the Windows Telephony environment.

lpfnLineCreateProc

Specifies a far pointer to the **LINEEVENT** callback procedure supplied by TAPI.DLL. The service provider will use this function to send **LINE_CREATE** messages when a new line device needs to be created. This function should not be called to send a **LINE_CREATE** message until after the service provider has returned from the **TSPI_providerInit** procedure.

lpfnPhoneCreateProc

Specifies a far pointer to the **PHONEEVENT** callback procedure supplied by TAPI.DLL. The service provider will use this function to send **PHONE_CREATE** messages when a new phone device needs to be created. This function should not be called to send a **PHONE_CREATE** message until after the service provider has returned from the **TSPI_providerInit** procedure.

In Windows Telephony version 1.0 (for Windows 3.1), TAPI.DLL examine the [Provider<ppid>] section of the TELEPHON.INI file to determine the number of lines and phones supported by the service provider. This function was added to support service providers with Plug'n'Play capability to determine the number of devices at run time without writing the number to a parameter file.

For backward compatibility, if TAPI.DLL attempts to invoke this function and the function is found to not be declared in the service provider, it will attempt to obtain the information from TELEPHON.INI (for backward compatibility). If the information is not found there, or if TSPI_providerEnumDevices fails, the service provider will be assumed to support no lines and no phones.

TSPI_providerCreateLineDevice

[New - Windows 95]

```
LONG TSPIAPI TSPI_providerCreateLineDevice (DWORD dwTempID,
    DWORD dwDeviceID)
```

This provider function is called by TAPI.DLL in response to receipt of a **LINE_CREATE** message from the service provider, which allows the dynamic creation of a new line device.

- Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

  LINEERR_BADDEVICEID

  LINEERR_NOMEM

  LINEERR_OPERATIONFAILED

This function must be declared in the DEF file of the service provider with the ordinal value 598.

dwTempID

Specifies the temporary device ID which the service provider passed to TAPI.DLL in the **LINE_CREATE** message.

dwDeviceID

Specifies the device ID that TAPI.DLL will assign to this device if this function is successful.

When TAPI.DLL receives a **LINE_CREATE** message from a service provider, it calls this function (it never calls this function spontaneously). TAPI.DLL adds 1 to the number of devices of that type, and passes the resulting new, unused device ID as the *dwDeviceID* parameter to this function. It also passes in the function the *dwParam2* from the **LINE_CREATE** message as *dwTempID*. Note that adding the new device to the end of the device list is likely to produce non-contiguous device IDs for the service provider; service providers that support dynamic device creation must also support non-contiguous device IDs (TAPI.DLL currently handles this fine; it keeps a separate pointer in each *tLine* to its associated driver info, so they don't have to be contiguous).

If the service provider recognizes the *dwTempID* and is successful in setting up the structures and such that it needs to support the new device, it saves off the *dwDeviceID*, and returns SUCCESS. If this function is unsuccessful, TAPI.DLL doesn't add the device, and there are no negative affects (the **LINE_CREATE** message is ignored). If this function completes successfully, TAPI.DLL saves the new number of devices, and creates a new entry in the device table (reallocating memory to expand the table). It then informs applications of the availability of the new device using **LINE_CREATE** or **LINE_LINEDEVSTATE** (LINEDEVSTATE_REINIT) messages.

For backward compatibility, older service providers will not export this function. However, they also should not send LINE_CREATE messages, which means TAPI.DLL would not try to call this function. In the unlikely event a LINE_CREATE message is received from a provider, and the provider is found to not export this function, the LINE_CREATE message is discarded and ignored.

TSPI_providerCreatePhoneDevice

[New - Windows 95]

```
LONG TSPIAPI TSPI_providerCreatePhoneDevice(DWORD dwTempID,
    DWORD dwDeviceID)
```

This provider function is called by TAPI.DLL in response to receipt of a **PHONE_CREATE** message from the service provider, which allows the dynamic creation of a new phone device.

- Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

  LINEERR_BADDEVICEID

  LINEERR_NOMEM

  LINEERR_OPERATIONFAILED

This function must be declared in the DEF file of the service provider with the ordinal value 599.

dwTempID

Specifies the temporary device ID which the service provider passed to TAPI.DLL in the **PHONE_CREATE** message.

dwDeviceID

Specifies the device ID that TAPI.DLL will assign to this device if this function is successful.

When TAPI.DLL receives a **PHONE_CREATE** message from a service provider, it calls this function (it never calls this function spontaneously). TAPI.DLL adds 1 to the number of devices of that type, and passes the resulting new, unused device ID as the *dwDeviceID* parameter to this function. It also passes in the function the *dwParam2* from the **PHONE_CREATE** message as *dwTempID*. Note that adding the new device to the end of the device list is likely to produce non-contiguous device IDs for the service provider; service providers that support dynamic device creation must also support non-contiguous device IDs (TAPI.DLL currently handles this fine; it keeps a separate pointer in each *tPhone* to its associated driver info, so they don't have to be contiguous).

If the service provider recognizes the *dwTempID* and is successful in setting up the structures and such that it needs to support the new device, it saves off the *dwDeviceID*, and returns SUCCESS. If this function is unsuccessful, TAPI.DLL doesn't add the device, and there are no negative affects (the **PHONE_CREATE** message is ignored). If this function completes successfully, TAPI.DLL saves the new number of devices, and creates a new entry in the device table (reallocating memory to expand the table). It then informs applications of the availability of the new device using **PHONE_CREATE** or **PHONE_STATE** (PHONESTATE_REINIT) messages.

For backward compatibility, older service providers will not export this function. However, they also should not send PHONE_CREATE messages, which means TAPI.DLL would not try to call this function. In the unlikely event a PHONE_CREATE message is received from a provider, and the provider is found to not export this function, the PHONE_CREATE message is discarded and ignored.

TSPI_lineDropOnClose

[New - Windows 95]

```
LONG TSPIAPI TSPI_lineDropOnClose (HDRVCALL hdCall)
```

This function must be declared in the DEF file of the service provider with the ordinal value 596.

- Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

  LINEERR_NOMEM

  LINEERR_OPERATIONFAILED

hdCall

Specifies the service provider's handle to the call to be dropped or turned over to control by external equipment. The call state of *hdCall* can be any state.

TAPI.DLL will call this function for each call owned by an application that calls the TAPI **lineClose** function, if, at the time **lineClose** is called, that application is the sole owner of the call.

A service provider which implements **TSPI_lineDropOnClose** shall dispose of the call according to the setting of the LINEDEVCAPFLAGS_CLOSEDROP bit in the dwDevCapFlags field in the LINEDEVCAPS structure returned by **TSPI_lineGetDevCaps**. If this bit is 1, the service provider will drop calls upon which this function is invoked. If this bit is 0, the service provider will not drop calls upon which this function is invoked, but will turn control of the calls over to an external phone or other equipment.

A service provider should be designed to drop calls upon invocation of **TSPI_lineDropOnClose** or **TSPI_lineClose** if, when no application remains as an owner of the call, there would be no external means of terminating the call (e.g., by use of physical buttons or hookswitches on a phone on the desktop).

A service provider which does not drop the call but instead turns it over to the control of external equipment may choose to "detach" the call from the API by immediately transitioning the call to the *idle* state (i.e., send a LINE_CALLSTATE message indicating LINECALLSTATE_IDLE) so that monitoring applications will know to call **lineDeallocateCall** and release their call handles. A call which is so "detached" would be *idle* from the point of view of the Telephony API even though it continues to be active and under the control of external equipment. Alternatively, the service provider can continue to present call state messages for the benefit of monitoring applications, up until the time the **TSPI_lineCloseCall** or **TSPI_lineClose** function is invoked.

Note that this function returns synchronously. If the service providers requires lengthy processing to drop the call (as with **TSPI_lineDrop**), the actual processing may be done asynchronously, but the function should return immediately indicating that the provider has accepted the request. Eventual completion of the drop or detach operation would be indicated by a LINE_CALLSTATE (*idle*) message.

For backward compatibility, if a service provider does not implement **TSPI_lineDropOnClose**, TAPI.DLL will instead call **TSPI_lineDrop** on all calls for which an application is the sole owner when that application calls **lineClose** on the line on which those calls exist. This behavior provides for backward compatibility with the previous version of TAPI.

The LINEDEVCAPFLAGS_CLOSEDROP bit indicates whether or not calls will be forcibly dropped by the service provider when a line is closed. The CLOSEDROP flag would not previously come into play, however, when the last owner of a call closed the line but one or more monitors continued to have a handle to the line, because **TSPI_lineClose** is not called until all applications (owners *and* monitors) close the line. In order to prevent calls remaining active without owners, the previous version of TAPI.DLL insisted on "cleaning up" active calls by calling **TSPI_lineDrop** on all calls for which the application that called **lineClose** was the sole owner. This behavior rendered the CLOSEDROP bit simply "advisory" to applications that some external equipment (such as an extension phone) might keep dropped calls active, since service providers had no way of knowing if **TSPI_lineDrop** had been explicitly called by the application or implicitly by TAPI.DLL.

This new function was create to accommodate environments in which hardware permits calls to remain active and under the control of external devices after all telephony applications quit and TAPI shuts down, and to allow providers to know whether the request to drop a call was made by an application

(user) or implicitly by TAPI.DLL.

TSPI_lineDropNoOwner

[New - Windows 95]

```
LONG TSPIAPI TSPI_lineDropNoOwner(HDRVCALL hdCall)
```

This function must be declared in the DEF file of the service provider with the ordinal value 597.

- Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

  LINEERR_NOMEM

  LINEERR_OPERATIONFAILED

hdCall

Specifies the service provider's handle to the call to be dropped or turned over to control by external equipment. The call state of *hdCall* can be any state.

TAPI.DLL will call this function when a new call is delivered by a service provider to TAPI.DLL (via a LINE_NEWCALL message followed by an initial LINE_CALLSTATE message) if no application can be found to become an owner of the call, if there are one or more monitoring applications active on the line, and if the call is in a state other than *idle* or *offering*. This prevents calls from being present in the system for which no application has responsibility to drop the call.

A service provider should be designed to drop calls upon invocation of **TSPI_lineDropNoOwner** if, when no application can be found to be an owner of the call, there would be no external means of terminating the call (e.g., by use of physical buttons or hookswitches on a phone on the desktop).

A service provider which does not drop the call but instead turns it over to the control of external equipment may choose to "detach" the call from the API by immediately transitioning the call to the *idle* state (i.e., send a LINE_CALLSTATE message indicating LINECALLSTATE_IDLE) so that monitoring applications will know to call **lineDeallocateCall** and release their call handles. A call which is so "detached" would be *idle* from the point of view of the Telephony API even though it continues to be active and under the control of external equipment. Alternatively, the service provider can continue to present call state messages for the benefit of monitoring applications, up until the time the **TSPI_lineCloseCall** or **TSPI_lineClose** function is invoked.

Note that this function returns synchronously. If the service providers requires lengthy processing to drop the call (as with **TSPI_lineDrop**), the actual processing may be done asynchronously, but the function should return immediately indicating that the provider has accepted the request. Eventual completion of the drop or detach operation would be indicated by a LINE_CALLSTATE (*idle*) message.

For backward compatibility, if a service provider does not implement **TSPI_lineDropNoOwner**, TAPI.DLL will instead call **TSPI_lineDrop** on all calls delivered in other then offering state for which an initial owner cannot be found. This behavior provides for backward compatibility with the previous version of TAPI.

TSPI_lineSetCurrentLocation

[New - Windows 95]

```
LONG TSPIAPI TSPI_lineSetCurrentLocation(DWORD dwLocation)
```

This function is called by TAPI.DLL whenever the address translation location is changed by the user (in the Dial Helper dialog) or an application (using the **lineSetCurrentLocation** function). Service providers which store parameters specific to a location (e.g., touch-tone sequences to invoke particular PBX functions) would use the location to select the set of parameters applicable to the new location.

- Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

  LINEERR_NOMEM

  LINEERR_OPERATIONFAILED

This function must be declared in the DEF file of the service provider with the ordinal value 600.

dwLocation

Specifies the permanent location ID of the selected location.

Service providers which store sets of parameters based on the user's current location should identify those sets by the permanent location ID of the location.

If the service provider does not implement specific parameters by location, this function can be omitted.

If the value passed in to the service provider by this function is not recognized by the service provider (for example, because no location-specific parameters have yet been set for the specified location), the service provider should use a default set of parameters.

Note that TAPI.DLL ignores the value returned by the service provider for this function; it is returned for debugging purposes only.

The service provider can synchronize its stored location-specific parameters with the locations stored by TAPI.DLL whenever the **TSPI_providerConfig** or **TSPI_lineConfigDialog** function is called, by using the **lineGetTranslateCaps** function to obtain a table of current locations. Newly-defined locations and removed locations can be detected at this time, and appropriate adjustments made to the service provider's stored location parameters (including prompting the user to make settings for new locations).

For backward compatibility, if the service provider does not export this function, TAPI.DLL will not call it, and no ill effects will result.

TSPI_lineConfigDialogEdit

[New - Windows 95]

```
LONG TSPI_lineConfigDialogEdit(DWORD dwDeviceID, HWND hwndOwner,
    LPCSTR lpszDeviceClass, LPVOID const lpDeviceConfigIn, DWORD dwSize,
    LPVARSTRING lpDeviceConfigOut)
```

This function causes the provider of the specified line device to display a modal dialog as a child window of *hwndOwner* to allow the user to configure parameters related to the line device.

- Returns zero if the request is successful or a negative error number if an error has occurred. Possible return values are:

  LINEERR_INVALDEVICECLASS LINEERR_OPERATIONFAILED
  LINEERR_INVALPARAM LINEERR_RESOURCEUNAVAIL
  LINEERR_NODRIVER LINEERR_OPERATIONUNAVAIL
  LINEERR_NOMEM

This function must be declared in the DEF file of the service provider with the ordinal value 601.

dwDeviceID
Specifies the line device to be configured.

hwndOwner
Specifies a handle to a window to which the dialog is to be attached.

lpszDeviceClass
Specifies a far pointer to a NULL-terminated string that identifies a device class name. This device class allows the caller to select a specific subscreen of configuration information applicable to that device class. If this parameter is NULL or points to an empty string, the highest level configuration should be selected. The permitted strings are the same as for **TSPI_lineGetID**.

lpDeviceConfigIn
Specifies a far pointer to the opaque configuration data structure that was returned by **TSPI_lineGetDevConfig** (or a previous invocation of **TSPI_lineConfigDialogEdit**) in the variable portion of the **VARSTRING** structure.

dwSize
Specifies the number of bytes in the structure pointed to by *lpDeviceConfigIn*. This value will have been returned in the **dwStringSize** field in the **VARSTRING** structure returned by **TSPI_lineGetDevConfig** or a previous invocation of **TSPI_lineConfigDialogEdit**.

lpDeviceConfigOut
Specifies a far pointer to the memory location of type **VARSTRING** where the device configuration structure is returned. Upon successful completion of the request, this location is filled with the device configuration. The **dwStringFormat** field in the **VARSTRING** structure will be set to **STRINGFORMAT_BINARY**. Prior to calling **lineGetDevConfig** (or a future invocation of **lineConfigDialogEdit**), the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI.DLL for returning information.

This function causes the service provider to display a modal dialog (attached to *hwndOwner*) to allow the user to configure parameters related to the line specified by *dwDeviceID*.

The *lpszDeviceClass* parameter selects a specific subscreen of configuration information applicable to the device class in which the user is interested; the permitted strings are the same as for **TSPI_lineGetID**. For example, if the line supports the Comm API, passing “COMM” as *lpszDeviceClass* causes the provider to display the parameters related specifically to Comm (or, at least, start at the corresponding point in a multilevel configuration dialog chain, so the user doesn’t have to “dig” to find the parameters of interest).

The *lpszDeviceClass* parameter would be “tapi/line” , “”, or NULL to cause the provider to display the highest level configuration for the line.

The difference between this function and **TSPI_lineConfigDialog** is the source of the parameters to edit and the result of the editing. In **TSPI_lineConfigDialog**, the parameters edited are those currently in use on the device (or set for use on the next call), and any changes made have (to the maximum

extent possible) an immediate impact on any active connection; also, the application must use **lineGetDevConfig** to fetch the result of parameter changes from **TSPI_lineConfigDialog**. With **TSPI_lineConfigDialogEdit**, the parameters to edit are passed in from the application, and the results are returned to the application, with *no* impact on active connections; the results of the editing are returned with this function, and the application does not need to call **lineGetDevConfig**. Thus, **TSPI_lineConfigDialogEdit** permits an application to provide the ability for the user to set up parameters for future calls without having an impact on any active call. Note, however, the output of this function can be passed to **TSPI_lineSetDevConfig** to affect the current call or next call.

For backward compatibility, this function will not be exported by older service providers. TAPI.DLL will detect this condition and report LINEERR_OPERATIONUNAVAIL should an application attempt to call this function on an older provider.

TSPI_lineReleaseUserUserInfo

[New - Windows 95]

```
LONG TSPI_lineReleaseUserUserInfo (DRV_REQUESTID dwRequestID,
    HDRVCALL hdCall)
```

This function informs the service provider that the user-user information contained in the LINECALLINFO structure has been processed, and that subsequently received user-user information can now be written into that structure. The service provider sends a LINE_CALLINFO message indicating LINECALLINFOSTATE_USERUSERINFO when new information is available.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_OPERATIONFAILED
LINEERR_NOMEM	LINEERR_RESOURCEUNAVAIL

This function must be declared in the DEF file of the service provider with the ordinal value 602.

dwRequestID

Specifies the identifier of the asynchronous request.

hdCall

Specifies the service provider's handle to the call for which user-user information is to be released. The call state of *hdCall* can be *any*.

The **TSPI_lineReleaseUserUserInfo** function permits control of the flow of incoming user-user information on an ISDN connection. When a new, complete user-user information message is received, the service provider informs TAPI.DLL using a **LINE_CALLINFO** message (specifying LINECALLINFOSTATE_USERUSERINFO). The user-user information and other fields in **LINECALLINFO** may be examined by multiple calls to **TSPI_lineGetCallInfo**. The service provider shall not overwrite previous user-user information in **LINECALLINFO** with newer information until after **TSPI_lineReleaseUserUserInfo** has been called. It is the responsibility of the service provider to buffer subsequently received user-user information until the previous information is released. Any remaining buffered information may be discarded when **TSPI_lineCloseCall** is invoked.

If this function is invoked while there is no user-user information in LINECALLINFO, the service provider should nevertheless return an indication of success.

For backward compatibility, TAPI.DLL will automatically return LINEERR_OPERATIONUNAVAIL if this function is invoked for a call on a line under the control of a service provider which does not export the function.

Messages

- LINE_CREATE
- PHONE_CREATE
- LINE_ADDRESSSTATE
- LINE_CALLSTATE
- LINE_LINEDEVSTATE
- PHONE_STATE

LINE_CREATE

[New - Windows 95]

```
htLine = (HTAPILINE) 0;
htCall = (HTAPICALL) 0;
dwMsg = (DWORD) LINE_CREATE;
dwParam1 = (DWORD) hProvider;
dwParam2 = (DWORD) TempDeviceId;
dwParam3 = (DWORD) 0;
```

A service provider sends a LINE_CREATE message to the **LINEEVENT** callback when it desires to create a new device. This can occur when a new device is detected by Plug'n'Play, or dynamically enabled by the user through a provider configuration function. across the TSPI. During the LINEEVENT callback procedure, TAPI.DLL simply queues this message for later processing. When the message is received from the queue, the **TSPI_providerCreateLineDevice** function will be called in the context of the TAPIEXE.EXE process.

- No return value.

htLine

Unused

htCall

Unused

dwMsg

The value LINE_CREATE

dwParam1

Specifies the service provider handle (*hProvider*) as received in **TSPI_providerInit**.

dwParam2

Contains a *temporary* device ID generated by the service provider. TAPI.DLL will use this value in a subsequent call to **TSPI_providerCreateLineDevice** to assist the provider in associating the message and the function call (in the event that multiple devices are being created).

dwParam3

Unused.

Service providers may (and should) continue to make "static" device allocations at startup time when TAPI.DLL calls **TSPI_providerEnumDevices**. Creating known devices using this mechanism, instead of always using **LINE_CREATE**, involves lower overhead for applications (since they don't have to process **LINE_CREATE** messages, updated device information, etc.). The **LINE_CREATE** mechanism is intended to be used only if new devices are created while the service provider is active (i.e., between **TSPI_providerInit** and **TSPI_providerShutdown**).

This message is sent to the **LINEEVENT** callback entry point in TAPI.DLL. The service provider receives a pointer to this callback in the **TSPI_providerEnumDevices** function and in each **TSPI_lineOpen** function; the **LINE_CREATE** message can be sent to the **LINEEVENT** callback function given to any open line or at startup.

Devices cannot be removed dynamically. If a service provider desires to remove a line from service, it would send a **LINE_LINEDEVSTATE** (LINEDEVSTATE_OUTOFSERVICE) message. It would then refuse (e.g., by returning LINEERR_INVALLINESTATE) to perform any operation with the device until it returns to service or the provider is shutdown and restarted (in which case the device could be not declared in **TSPI_providerEnumDevices**, effectively removing it from the system).

For backward compatibility, older service providers would not be expected to send this message. If they do, the message should be treated in the same manner as described above for new service providers.

PHONE_CREATE

[New - Windows 95]

```
htPhone = (HTAPIPHONE) 0;
dwMsg = (DWORD) PHONE_CREATE;
dwParam1 = (DWORD) hProvider;
dwParam2 = (DWORD) TempDeviceId;
dwParam3 = (DWORD) 0;
```

A service provider sends a PHONE_CREATE message to the **PHONEEVENT** callback when it desires to create a new device. This can occur when a new device is detected by Plug'n'Play, or dynamically enabled by the user through a provider configuration function. across the TSPI. During the PHONEEVENT callback procedure, TAPI.DLL simply queues this message for later processing. When the message is received from the queue, the **TSPI_providerCreatePhoneDevice** function will be called in the context of the TAPIEXE.EXE process.

- No return value.

htPhone

Unused

dwMsg

The value PHONE_CREATE

dwParam1

Specifies the service provider handle (*hProvider*) as received in **TSPI_providerInit**.

dwParam2

Contains a *temporary* device ID generated by the service provider. TAPI.DLL will use this value in a subsequent call to **TSPI_providerCreatePhoneDevice** to assist the provider in associating the message and the function call (in the event that multiple devices are being created).

dwParam3

Unused.

Service providers may (and should) continue to make "static" device allocations at startup time when TAPI.DLL calls **TSPI_providerEnumDevices**. Creating known devices using this mechanism, instead of always using **PHONE_CREATE**, involves lower overhead for applications (since they don't have to process **PHONE_CREATE** messages, updated device information, etc.). The **PHONE_CREATE** mechanism is intended to be used only if new devices are created while the service provider is active (i.e., between **TSPI_providerInit** and **TSPI_providerShutdown**).

This message is sent to the **PHONEEVENT** callback entry point in TAPI.DLL. The service provider receives a pointer to this callback in the **TSPI_providerEnumDevices** function and in each **TSPI_phoneOpen** function; the **PHONE_CREATE** message can be sent to the **PHONEEVENT** callback function given to any open phone or at startup.

Devices cannot be removed dynamically. If a service provider desires to remove a phone from service, it would send **PHONE_STATE** (PHONESTATE_DISCONNECTED) message. It would then refuse (e.g., by returning PHONEERR_INVALPHONESTATE) to perform any operation with the device until it returns to service or the provider is shutdown and restarted (in which case the device could be not declared in **TSPI_providerEnumDevices**, effectively removing it from the system).

For backward compatibility, older service providers would not be expected to send this message. If they do, the message should be treated in the same manner as described above for new service providers.

LINE_ADDRESSSTATE

[New - Windows 95]

Add the following to the description of the *dwParam2* parameter:

LINEADDRESSSTATE_CAPSCHANGE

Indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the **LINEADDRESSCAPS** structure for the address have changed. If a service provider sends a **LINE_ADDRESSSTATE** message containing this value to TAPI.DLL, TAPI will pass it along to applications which have negotiated this or a subsequent API version; applications negotiating a previous API version will receive **LINE_LINEDEVSTATE** messages specifying LINEDEVSTATE_REINIT, requiring them to shutdown and reinitialize their connection to TAPI.DLL in order to obtain the updated information.

For backward compatibility, older service providers would not be expected to generate this value in a LINE_ADDRESSSTATE message. If they do, they message should be handled in the same manner as for newer service providers (as described above).

LINE_CALLSTATE

[New - Windows 95]

Add the following to the description of the *dwParam2* parameter:

> If *dwParam1* is **LINECALLSTATE_CONFERENCED**, *dwParam2* contains the *htCall* of the parent call of the conference of which the subject *htCall* is a member. If the call specified in *dwParam2* was not previously considered by TAPI.DLL to be a parent conference call, this message causes it to be so treated. The call specified in *dwParam2* must already exist; it was most likely previously created by a **LINE_NEWCALL** message and set to **LINECALLSTATE_ONHOLDPENDCONF**.

For backward compatibility, older service providers will not pass a valid *htCall* in *dwParam2*. TAPI.DLL must check the value passed, and ignore it if it is not a calid *htCall*. If the value is a valid *htCall*, TAPI.DLL will also check the API version in use on the line device, and establish a conference call internally only if the API version is 0x00010004 or greater (i.e., if the API version on the line is older the 0x00010004, this parameter should be ignored).

LINE_LINEDEVSTATE

[New - Windows 95]

Add the following to the description of the *dwParam1* parameter:

LINEDEVSTATE_CAPSCHANGE

Indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the **LINEDEVCAPS** structure for the address have changed. If a service provider sends a **LINE_LINEDEVSTATE** message containing this value to TAPI.DLL, TAPI will pass it along to applications which have negotiated this or a subsequent API version; applications negotiating a previous API version will receive **LINE_LINEDEVSTATE** messages specifying LINEDEVSTATE_REINIT, requiring them to shutdown and reinitialize their connection to TAPI.DLL in order to obtain the updated information.

LINEDEVSTATE_CONFIGCHANGE

Indicates that configuration changes have been made to one or more of the media devices associated with the line device. If a service provider sends a **LINE_LINEDEVSTATE** message containing this value to TAPI.DLL, TAPI will pass it along to applications which have negotiated this or a subsequent API version; applications negotiating a previous API version will not receive any notification.

LINEDEVSTATE_COMPLCANCEL

Indicates that the call completion identified by the completion ID contained in parameter *dwParam2* of the **LINE_LINEDEVSTATE** message has been externally cancelled and is no longer considered valid (if that value were to be passed in a subsequent call to **lineUncompleteCall**, the function would fail with **LINEERR_INVALCOMPLETIONID**). If a service provider sends a **LINE_LINEDEVSTATE** message containing this value to TAPI.DLL, TAPI will pass it along to applications which have negotiated this or a subsequent API version; applications negotiating a previous API version will not receive any notification.

For backward compatibility, older service providers would not be expected to generate these values. If they do, TAPI.DLL will treat them the same as if the service provider were using API version 0x00010004 or greater (i.e., as described above).

PHONE_STATE

[New - Windows 95]

Add the following to the description of the *dwParam1* parameter:

PHONESTATE_CAPSCHANGE

Indicates that, due to configuration changes made by the user or other circumstances, one or more of the fields in the **PHONECAPS** structure have changed. If a service provider sends a **PHONE_STATE** message containing this value to TAPI.DLL, TAPI will pass it along to applications which have negotiated this or a subsequent API version; applications negotiating a previous API version will receive **PHONE_STATE** messages specifying PHONESTATE_REINIT, requiring them to shutdown and reinitialize their connection to TAPI.DLL in order to obtain the updated information.

For backward compatibility, older service providers would not be expected to generate this value in a PHONE_STATE message. If they do, they message should be handled in the same manner as for newer service providers (as described above).

TSPI Programmer's Guide Corrections and Updates

TSPI Programmer's Guide Corrections and Updates

TSPI Programmer's Guide Corrections and Updates

General

Asynchronous Phone Operations

There are the nine asynchronous operations related to phones. These are initiated by the **TSPI_phoneDevSpecific**, **TSPI_phoneSetButtonInfo**, **TSPI_phoneSetData**, **TSPI_phoneSetDisplay**, **TSPI_phoneSetGain**, **TSPI_phoneSetHookSwitch**, **TSPI_phoneSetLamp**, **TSPI_phoneSetRing**, and **TSPI_phoneSetVolume** functions. If an application calls the **phoneClose** function and has the only handle to the phone, TAPI calls the **TSPI_phoneClose** function to direct the service provider to terminate the asynchronous operations and call the **ASYNC_COMPLETION** callback function as appropriate. If the application does not have the only a handle to the phone, the service provider receives no notification of the request and any outstanding asynchronous operations remain active.

Call-less Asynchronous Operations

There are five asynchronous operations which are not related to any particular call. These are initiated by the **lineDevSpecific**, **lineDevSpecificFeature**, **lineForward**, **lineSetTerminal**, and **lineUncompleteCall** functions. If an application has initiated one of these asynchronous operations and calls **lineClose**, the service provider may receive no indication that the application has abandoned the function. This would occur if the application was not the only application to have the line open. If the application calls **lineClose** with one of these operations pending and it is the only application with a handle to the line, TAPI calls the **TSPI_lineClose** function and the service provider should termination the asynchronous operation, calling the **ASYNC_COMPLETION** callback function for each such outstanding operation with the LINEERR_OPERATIONFAILED error value.

Capabilities Bits

After a service provider has set capability bits in the **LINEDEVCAPS**, **LINEADDRESSCAPS**, **LINECALLINFO**, and **PHONECAPS** to indicate the states and messages it supports, TAPI automatically sets additional bits to indicate the messages and states it supports idependently of the service provider. The following lists the capabilities bits TAPI sets:

In **lineGetDevCaps**:

```
LINEDEVCAPS.dwLineStates |=
    LINEDEVSTATE_OPEN |
    LINEDEVSTATE_CLOSE |
    LINEDEVSTATE_REINIT |
    LINEDEVSTATE_TRANSLATECHANGE;
```

In **lineGetAddressCaps**:

```
LINEADDRESSCAPS.dwCallInfoStates |=
    LINECALLINFOSTATE_NUMOWNERINCR |
    LINECALLINFOSTATE_NUMOWNERDECR |
    LINECALLINFOSTATE_NUMMONITORS;

LINEADDRESSCAPS.dwCallStates |= LINECALLSTATE_UNKNOWN;
```

In **lineGetCallInfo**:

```
LINECALLINFO.dwCallStates |= LINECALLSTATE_UNKNOWN;
```

In **phoneGetDevCaps**:

```
PHONECAPS.dwPhoneStates |=
    PHONESTATE_OWNER |
    PHONESTATE_MONITORS |
    PHONESTATE_REINIT;
```

Child Conference Calls

Service providers should remove conference children from the *conferenced* state (either *idle* or some other call state) prior to putting the conference parent in LINECALLSTATE_IDLE. Otherwise, the children remain in the linked list of conference children, causing problems when TAPI closes the conference in response to a call to the **TSPI_lineCloseCall** function for the parent conference call.

Externally Generated Phone Numbers

A service provider can indicate externally generated phone numbers (that is, numbers for outbound calls generated external to the computer) by setting the DisplayableAddress, CalledParty, or Comment members in the **LINECALLINFO** structure when processing the **TSPI_lineGetCallInfo** function. If TAPI has not save its own data for these members, it leaves the data set by the service provider unchanged. If TAPI does have cached data for these members, TAPI adds its text to the end of the variable portion of the **LINECALLINFO** structure and overwrites the size and offset put by the service provider into the fixed portion.

A service provider can also copy an outbound number into the CalledID member. Therefore, logging applications should check the CalledID member on outbound calls if the DisplayableAddress and CalledParty members are empty.

EXTIDGEN Utility

The Extidgen utilities does not generate extension identifiers unless the computer on which it is run is also running NetBios or other network software.

Filling Variable-Length Structures

When filling variable-length structures, the service provider should always start immediately after the end of the fixed portion of the structure and leave any extra space at the end of the allocated memory so that TAPI can use it for the variable-length members it is responsible for.

Idle Call State

If a service provider changes the call state to *idle* , it must *not* subsequently change the call to any other state. The *idle* state should be the last state a call enters and it should remain in that state until **TSPI_lineCloseCall** is called to dispose of it.

Pending Call Handles

A service provider is required to create one or more call handles (variables of type HDRVCALL) whenever TAPI calls one of these functions: **TSPI_lineMakeCall**, **TSPI_lineCompleteTransfer**, **TSPI_lineForward**, **TSPI_linePickup**, **TSPI_linePrepareAddToConference**, **TSPI_lineSetupConference**, **TSPI_lineSetupTransfer**, and **TSPI_lineUnpark**. When a service provider receives such a call, the service provider must set the variable supplied by TAPI to the value of the handle *before* returning from the call. TAPI considers this “pending call handle” to be tentatively valid. When the service provider actually creates the call, it must call the **ASYNC_COMPLETION** callback to formally validate the handle.

If TAPI calls the **TSPI_lineCloseCall** function before the service provider has formally validated a pending call handle, the service provider must stop all further processing associated with the call handle and call the **ASYNC_COMPLETION** callback function using the error value LINEERR_OPERATIONFAILED.

Protected-Mode Operation

Service providers are intended to be run in protected mode. Therefore, the PROTMODE flag must be given in service provider module definition file (.DEF) to ensure proper operation.

Termination of Asynchronous Operations

When an application calls the **lineClose** function with one or more asynchronous operations outstanding, TAPI may direct the service provider to termination asynchronous operations assocated with a call, depending on whether the application is the sole owner of the call and has the only handle to the call.

If the application is the sole owner of a call, TAPI calls **TSPI_lineDropOnClose** or **TSPI_lineDrop** (depending on the provider) for each such call. The service provider should check for any pending asynchronous operations associated with the call being dropped. If a pending operation exists, the service provider should take the appropriate action, possibly terminating the operation in progress.

If the application has the only handle to a call (that is, there are no other owners or monitors), TAPI calls the **TSPI_lineCloseCall** function. The service provider must consider this call to be the absolute indication that pending asynchronous operations must be abandoned. The service provider must ensure an orderly completion of the call and must call the **ASYNC_COMPLETION** callback function for all outstanding asynchronous operations, specifying the LINEERR_OPERATIONFAILED error value. If TAPI previously called the **TSPI_lineDropOnClose** or **TSPI_lineDrop** function, it calls **TSPI_lineCloseCall** immediately after the service provider returns from the other call; it does not wait for the asynchronous operation associated with the **TSPI_lineDrop** function to complete. The service provider must complete this outstanding operation and call the **ASYNC_COMPLETION** callback function.

If the application is not the sole owner of the call, TAPI does not call **TSPI_lineDropOnClose** or **TSPI_lineDrop**. If the application does not have the only handle to the call, TAPI does not call the **TSPI_lineCloseCall** function. If the application is neither the sole owner nor the sole possessor of a handle to the call, TAPI sends no notification to the service provider, and therefore any pending asynchronous operations remain intact. This means that such applications cannot terminate asynchronous operations they have started by using the **lineClose** function. However, because every asynchronous operation requires the invoking application to be an owner of the call, the likelihood of an application not being able to terminate asynchronous operations is very rare. If it does occur, the other owner(s) of the call(s) must take responsibility for the disposition of the call(s).

If TAPI calls **TSPI_lineDropOnClose** or **TSPI_lineDrop**, the service provider must eventually send a LINECALLSTATE_IDLE message for the associated call (unless **TSPI_lineCloseCall** is called first) so that monitors on the call can clean up. When the last monitor has called the **lineDeallocateCall** function, TAPI calls the **TSPI_lineCloseCall** function; any pending operations must be completed or terminated and ASYNC_COMPLETION called, as described above.

Transfer Data From Application Memory

Service providers should always copy input data from application memory into service provider memory before returning from an asynchronous function.

Unowned Calls

To prevent unowned calls from being in the telephony system, TAPI drops calls that have been signalled up from a service provider if it cannot identify an application to be the initial owner of the call. In Windows 3.1, TAPI calls the **TSPI_lineDrop** function to drop the call. In Windows 95, TAPI calls the **TSPI_lineDropNoOwner** function instead if the service provider exports the function.

The service provider can know in advance whether or not there is an application available to take ownership of a new call. TAPI always informs the service provider of the media modes for which there is an application available by using the **TSPI_lineSetDefaultMediaDetection** function. For example, if a service provider determines that there is an active call on the line having LINEMEDIAMODE_INTERACTIVEVOICE mode, it should check the default media detection before generating the LINE_NEWCALL and LINE_CALLSTATE messages. If the default media detection shows that no application is looking for a LINEMEDIAMODE_INTERACTIVEVOICE call, the service provider should not send the messages because TAPI will immediately drop it. If at some point, TAPI calls the **TSPI_lineSetDefaultMediaDetection** function to indicate that some application is available to take ownership of the call, the service provider can then send the LINE_NEWCALL and LINE_CALLSTATE messages and the call will not be dropped.

Functions, Messages, Structures

LINE_CLOSE

[New - Windows 95]

The LINE_CLOSE message is a *request* from the service provider to TAPI to close the line. The service provider must not actually close the line until TAPI acknowledges the request by calling the **TSPI_lineClose** function to direct the service provider to close the line.

LINECALLPARAMS

The last paragraph in the comments should read as follows:

Note that the parameters **DialParams** through **dwDevSpecificOffset** are ignored when an *lpCallParams* parameter is specified with the function **TSPI_lineConditionalMediaDetection**.

[New - Windows 95]

TSPI_lineCloseCall

[New - Windows 95]

TAPI does not call **TSPI_lineCloseCall** if an service provider synchronously returns an error from a call to the **TSPI_lineMakeCall** function. But TAPI does call **TSPI_lineCloseCall** if the service provider returns an error from the asynchronous operation initaietd by **TSPI_lineMakeCall**.

TSPI_lineCompleteTransfer

[New - Windows 95]

The service provider must set the variable supplied by TAPI to the value of the request call handle *before* returning from this function call.

TSPI_lineConditionalMediaDetection

[New - Windows 95]

The **TSPI_lineConditionalMediaDetection** function should check the bits set in the **dwCallParamFlags** member of the **LINECALLPARAMS** structure and handle the following cases as described:

The **TSPI_lineConditionalMediaDetection** function should return success if passing the same bit values to the **TSPI_lineMakeCall** function would also return success.

If the SECURE, ORIGOFFHOOK, and DESTOFFHOOK bits are set and the *dwAddressMode* parameter is LINEADDRESSMODE_ADDRESSID, the function should return success if it can succeed on one or more addresses on the line.

If the SECURE, ORIGOFFHOOK, and DESTOFFHOOK bits are set and the *dwAddressMode* parameter is LINEADDRESSMODE_DIALABLEADDR, the function should return success if it can succeed on the address identified by the *dwOrigAddress* parameter.

TSPI_lineDial

[New - Windows 95]

If the string pointed to by the *lpszDestAddress* parameter in the previous call to the **TSPI_lineMakeCall** or **TSPI_lineDial** function is terminated with a semicolon, an empty string in the current call to **TSPI_lineDial** indicates that dialing is complete.

TSPI_lineGatherDigits

[New - Windows 95]

When the operation associated with a call to the **TSPI_lineGatherDigits** function is cancelled (by a subsequent call to the function), the service provider should copy any digits collected up to that point to the buffer specified in the original call.

TSPI_lineGetAddressCaps

[New - Windows 95]

After the service provider returns from the **TSPI_lineGetAddressCaps** function, TAPI set the **dwCallInfoStates** and **dwCallStates** members of the **LINEADDRESSCAPS** structure as follows: :

```
LINEADDRESSCAPS.dwCallInfoStates |=
    LINECALLINFOSTATE_NUMOWNERINCR |
    LINECALLINFOSTATE_NUMOWNERDECR |
    LINECALLINFOSTATE_NUMMONITORS;

LINEADDRESSCAPS.dwCallStates |= LINECALLSTATE_UNKNOWN;
```

TSPI_lineGetCallInfo

[New - Windows 95]

After the service provider returns from the **TSPI_lineGetCallInfo** function, TAPI set the **dwCallStates** member of the **LINECALLINFO** structure as follows: :

```
LINECALLINFO.dwCallStates |= LINECALLSTATE_UNKNOWN;
```

TSPI_lineGetDevCaps

[New - Windows 95]

After the service provider returns from the **TSPI_lineGetDevCaps** function, TAPI set the **dwLinesStates** member of the **LINEDEVCAPS** structure as follows:

```
LINEDEVCAPS.dwLineStates |=
    LINEDEVSTATE_OPEN |
    LINEDEVSTATE_CLOSE |
    LINEDEVSTATE_REINIT |
    LINEDEVSTATE_TRANSLATECHANGE;
```

TSPI_lineGetDevConfig

[New - Windows 95]

If the **dwTotalSize** member of the **VARSTRING** structure pointed to by the *lpDeviceConfig* parameter is greater than or equal to the size of the fixed portion of the structure, the service provider should set the **dwNeededSize** member to the required size and return 0.

TSPI_lineGetID

[New - Windows 95]

The **TSPI_lineGetID** function returns LINEERR_NODEVICE.

TSPI_lineMonitorDigits

[New - Windows 95]

The **TSPI_lineMonitorDigits** function must return 0 when digit monitoring is cancelled (that is, when the *dwDigitModes* parameter is zero).

TSPI_lineOpen

[New - Windows 95]

The TSPI_lineOpen function does not return LINEERR_INVALMEDIAMODE.

TSPI_linePark

[New - Windows 95]

If the memory pointed to by the *lpNonDirAddress* parameter is not large enough for the requested address, the **TSPI_linePark** function should return LINEERR_STRUCTURETOOSMALL.

TSPI_lineSendUserUserInfo

[New - Windows 95]

The **TSPI_lineSendUserUserInfo** function can be also called when the call state is *offering*, *accepted*, or *ringback*.

TSPI_lineSetTerminal

[New - Windows 95]

Call progress tones and/or messages get routed to the same place as media. For example, if audio signals are going to the phone, then so will busy signals (analog) or Q.931 messages indicating busy (digital).

TSPI_lineSetupConference

[New - Windows 95]

The **TSPI_lineUnhold** function can recover calls that have the call state *onHoldPendingConference*. If this is done, any consultation call will typically go to the *idle* state.

TSPI_lineSetupTransfer

[New - Windows 95]

The **TSPI_lineUnhold** function can recover calls that have the call state *onHoldPendingTransfer*. If this is done, any consultation call will typically go to the *idle* state.

TSPI_lineUnhold

[New - Windows 95]

The **TSPI_lineUnhold** function should also recover calls that are on “soft hold”, that is, have the call state LINECALLSTATE_ONHOLDPENDINGTRANSFER or LINECALLSTATE_ONHOLDPENDINGCONFERENCE.

TSPI_phoneDevSpecific

[New - Windows 95]

The *lpParams* parameter of the **TSPI_phoneDevSpecific** function should not contain pointers. To get information back to the application from **TSPI_phoneDevSpecific** the service provide should send a PHONE_DEVSPECIFIC message with that information.

TSPI_phoneGetButtonInfo

The **TSPI_phoneGetButtonInfo** function returns the PHONEERR_NOMEM value if the service provider cannot access the memory containing the button information.

TSPI_phoneGetDevCaps

[New - Windows 95]

After the service provider returns from the **TSPI_phoneGetDevCaps** function, TAPI set the **dwPhoneStates** member of the **PHONECAPS** structure as follows: :

```
PHONECAPS.dwPhoneStates |=
    PHONESTATE_OWNER |
    PHONESTATE_MONITORS |
    PHONESTATE_REINIT;
```

TSPI_phoneGetExtensionID

The **TSPI_phoneGetExtensionID** function does not return the values PHONEERR_BADDEVICEID and PHONEERR_INVALPOINTER. TAPI never passes bad device identifiers or invalid pointers to the service provider.

TSPI_phoneGetID

The **TSPI_phoneGetID** function returns PHONEERR_INVALDEVICECLASS.

TSPI_providerEnumDevices

[New - Windows 95]

TAPI sets the variables pointed to by the *lpdwNumLines* and *lpdwNumPhones* parameters to either zero or the current value specified in the INI file. A service provider can update the values in these variable, if necessary, to specify the actual number of lines or phones supported.

If the service provider can not determine how many lines or phones it supports, it should set the corresponding variable to zero.

Telephony Services

About Telephony Services

Telephony is a technology that integrates computers with the telephone network. With telephony, people can use their computers to take advantage of a wide range of sophisticated communications features and services over a telephone line. The Windows Telephony service provider interface (TSPI) lets programmers develop back-end services that handle requests from applications to carry out and control communications over the telephone network.

Windows Telephony supports both speech and data transmission, allows for a variety of terminal devices, and supports complex connection types and call-management techniques such as conference calls, call waiting, and voice mail. Telephony allows all elements of telephone usage to be controlled by applications through the telephonty application programming interface (TAPI). These applications rely on the existence of service providers that support this functionality. The role of the service provider is to provide TSPI functions that carry out the tasks requested by applications using the TAPI functions.

Telephone Network Services

You can develop service providers for a variety of telphone network services and technologies, including:

- POTS (Plain Old Telephone Service). Voice and data transmitted in analog format while in the *local loop*, digitally elsewhere. Typically one media mode per call, one channel per line.
- ISDN (Integrated Services Digital Network). Transmitted digitally. Speeds of up to 128 Kbps on Basic Rate Interface (BRI-ISDN) lines and much higher on Primary Rate Interface (PRI-ISDN) lines. At least three channels and as many as 32 channels, for simultaneous, independently operated transmission of voice and data. International standard.
- T1/E1. Transmitted digitally.
- Switched 56. Signaling at 56 Kbps over dial-up telephone lines, but requires special equipment. Limited to calls to other specially equipped facilities.
- CENTREX. Centralized network services through regular telephone lines and using telephone-company equipment. No special equipment required.
- Digital PBXs (Private Branch Exchanges) and key systems.

Service Provider

A service provider is a dynamic link library that supports communications over a telephone network through a set of exported service functions. The service provider responds to telephony requests, sent to it by the TAPI dynamic link library, by carrying out the low-level tasks necessary to communicate over the telephone network. In this way, the service provider, in conjunction with TAPI, shield applications from the service- and technology-dependent details of the telephone network communication.

You write a service provider to extend telephony services for existing hardware or to provide telephony services for new hardware. Each service provider supports at least one hardware device, such as a fax board, an ISDN card, a telephone, or a modem. Some providers support a combination of devices. The installation utility or the user associates a service provider with its hardware device(s) when the service provider is installed. Multiple service providers can be installed as long as they access different devices.

Each service provider is responsible for responding to telephony requests from TAPI to control for line and phonedevices. A service provider is also responsible for controlling and accessing the information exchanged over a call. To manage this information (sometimes called the *media stream*), the service provider must provide additional capabilities or functions. For example, a service provider that supports fax or data transmission must implement the TSPI functions that control and monitor the line over which data is sent. It must either implement functions that carry out the actual transmission of the data over the line or ensure that applications can use existing Win32 functions to carry out the transmission.

If a service provider implements its own functions to manage the media stream, it is recommended that the functions be based on standard Win32 functions, such as the communication or multimedia functions. For nonstandard or custom hardware devices, however, it may be best to develop a custom interface.

To access the actual device, a device driver is required. Although, in some cases, a service provider may contain a device-driver component, it is recommended that service providers either use existing device drivers or develop the device driver as a separate, installable component.

The sample service provider, not intended to be a full-featured service provider, is the base on which you can develop more extensive service providers.

Models

Service providers can support different “models” of the ways lines and phones are handled based on how the devices are physically configured. The following paragraphs describe a selection of configurations that a service provider might support.

The *computer-centric* model uses a computer add-in card or an external add-on box that is connected to both the telephone network switch and the phone set. This model can easily integrate modem and fax functions, as well as the use of the telephone as an audio I/O device.

The *phone-centric* model connects the computer to the switch through the desktop phone set. Such phone sets typically connect to the computer through one of its serial ports. The service provider converts requests into telephony commands, such as commands based on the Hayes AT command set (ANSI/TIA/EIA-602), and sends them over the serial connection to the telephone. This configuration is limited because it generally provides only line control. There is no media-stream access to the computer.

A single BRI-ISDN line could fit both the phone-centric and computer-centric models. The service provider can treat this capability in a number of ways:

- Model the BRI line as a single line device with a pool of two channels, allowing both channels to be combined for establishing 128 Kbps calls.
- Model each B-channel as a separate line device and disallow the combination of the two channels into a single 128 Kbps channel.
- Model the BRI connection as two separate line devices, each drawing up to 2 channels from a shared pool of 2 B-channels.
- Model as three line devices: one for each of the two B-channels and one for the combination.

In the latter two models, channels may be assigned to different line devices at different times.

A LAN-based server might have multiple telephone-line connections to the switch. TAPI operations invoked at any of the client computers are forwarded over the LAN to the server. The server uses third-party call control between the server and the switch to implement the client’s call-control requests.

This model offers a lower cost per computer for call control if the LAN is already in use, and it also offers reduced cost for media stream access if shared devices such as voice digitizers, fax and/or data modems, and interactive voice response cards are installed in the server. The digitized media streams can be carried over the LAN, although real-time transfer of media may be problematic with some LAN technologies due to inconsistent throughput.

A LAN-based host can be connected to the switch using a switch-to-host link. TAPI operations invoked at any of the client computers are forwarded over the LAN to the host, which uses a third-party switch-to-host link protocol to implement the client’s call-control requests.

Note that it is also possible for a private branch exchange (PBX) to be directly connected to the LAN, and for the server functions to be integrated into the PBX. Within this model, different sub-configurations are possible:

- To provide personal telephony to each desktop, the service provider could model the PBX line associated with the computer (on a desktop) as a single line device with one channel. Each client computer would have one line device available.
- Each third-party station can be modeled as a separate line device to allow applications to control calls on other stations. (In a PBX, a *station* is anything to which a wire leads from the PBX). This enables the application to control calls on other stations. This solution requires that the application open each line it wants to manipulate or monitor, which may be satisfactory if only a small number of lines is of interest, but may generate excessive overhead if a large number of lines is involved.
- Model the set of all third-party stations as a single line device with one address (one phone number) assigned to it per station. Only a single device is to be opened, providing monitoring and control of all addresses (all stations) on the line. To originate a call on any of these stations, the application must only specify the station’s address to the function that makes the call. No extra **lineOpen** operations are required. However, this modeling implies that all stations have the same line-device capabilities, although their address capabilities could be different.

A potential advantage of this model is a lowered cost per computer if the LAN is already in use, but a limitation would be a possible lack of media-stream access by the computers.

The computer in use need not be a desktop computer. It can also be a laptop or other portable computer connected to the telephone network over a wireless connection.

The computer's connection may be shared by other telephony equipment. For an application to operate properly in this arrangement, neither the application nor the service provider can assume that there are no other active devices on the line.

Telephony Model

Telephony Model

Telephony Model

About Telephony Model

The service provider must implement and export the telephony service provider interface (TSPI) functions needed to ensure that the requests from the TAPI dynamic link library are carried out correctly.

The service provider must process all requests received from the TAPI dynamic link library.

A service provider does not have access other service providers.

Although TAPI grants simultaneous access by multiple applications to a single service provider, the service provider is not responsible for managing this access.

TAPI assigns a device identifier to each service provider. The set of device identifiers for all services providers is contiguous and non-overlapping.

The service provider supports the low-level device-specific aspects of telephony. This includes functions such as managing the states of calls, dialing, signal generation and detection, and switchboard controlling.

The implementations of such functions differ greatly from one device to another. For example, a service provider that provides direct management of analog phone lines would contain and control hardware for detecting on/off hook signals, generating and filtering tones, and so on. A service provider that worked in conjunction with a PBX would send command streams over a digital network to request similar operations from a piece of remote hardware. Service providers hide the device-specific details of how telephony operations are implemented.

Device Classes

There are two *device classes*: line and phone.

Each device has a corresponding set of TSPI functions and messages.

A line device is any device that has *line behavior*.

A phone is any device that has *phone behavior*.

The service provider implements this behavior by mapping requests into actual device commands.

The SPI can be mapped to a wide range of environments, including ones not traditionally thought of as telephony environments.

Phone devices and calls on line devices are capable of carrying media streams, such as voice, data, and video.

Line Device

Each line has one or more channels. The channels are identical and therefore interchangable.

The service provider can pool channels for use as shared resources.

The service provider can combine of two or more channels for use in a single call to provide a wider bandwidth for exchange of information. This technique is often called *inverse multiplexing*.

The service provider allocates one or more channels in a line in response to a request to make or answer a call.

Because channels are identical, the With a line's resources having identical capabilities and being interchangeable, there is no need for TAPI.DLL to explicitly identify which channels are to be used.

The service provider owns and completely manages the channels of each line.

Phone Devices

Service providers are not required to support phone devices.

Line and phone devices are *independent* devices.

Service providers that fully implement this independence can offer uses of the line and phone devices to the applications not defined by “normal” telephony protocols. For example, you can use the handset of the desktop’s phone as a wave form audio device for voice recording or playback, and, depending on the implementation, you may be able to do it without the switch knowing that the phone is in use. In such an implementation, lifting the local phone handset may not automatically send an offhook signal to the switch. This feature also allows your application to ring the local phone independent of incoming calls. Capabilities of actual service providers may be limited by the hardware and software used to interconnect the switch, the phone, and the computer. The SPI includes functions to retrieve device capabilities that allow clients to determine whether such a usage model is supported.

Call Objects

Call objects are dynamically created and have no external names. A call object represents the local endpoint of a connection to one or more remote parties. A call object can come into existence on demand (for outbound calls) or spontaneously (for inbound calls). Calls are strongly tied to the line on which they occur; functions to create calls take a line object as a parameter to indicate their parent line. Call objects are used as parameters to the functions that operate on calls including the operation to end the lifetime of a call. Calls remain subordinate to lines in several senses, for example, closing a line forces the calls on that line to be destroyed.

Service Levels

There are these levels of service:

Basic Telephony. Provides a minimum set of functions that corresponds to POTS. All service providers must support Basic Telephony.

Supplementary Telephony. Provides advanced switch features such as hold, transfer, and so on. All Supplementary services are optional; that is, the service provider is not required to support them.

Extended Telephony. Provides well-defined API extension mechanisms that enable application developers to access service provider-specific functions not directly defined by the Telephony API.

The set of TAPI functions do not map one-to-one onto the set of TSPI functions. In particular, functions related to privilege, phone number translation, and inter-application communication are handled by the TAPI.DLL and have no corresponding function in TSPI. Other functions such as those used for service-provider configuration and initialization have no corresponding functions in the TAPI.

Basic Telephony Services

All service providers must support Basic Telephony. Basic Telephony roughly corresponds to the features of POTS.

Phone-device services are optional.

Phone device services are *not* a part of Basic Telephony.

Supplementary Telephony Services

Supplementary telephony includes features found on modern PBXs, such as hold, transfer, conference, park, and so on. All supplementary features are optional.

A service provider provides any combination of the services.

A service provider specifies which services it supports through response to functions such as **TSPI_lineGetDevCaps** or **TSPI_lineGetAddressCaps**.

A single supplementary service may consist of multiple function calls and messages. It is important to point out that the Telephony API, and not the service provider developer, defines the behavior of each of these supplementary features.

A service provider should provide a supplementary telephony service only if it can implement the exact meaning as defined by the API. If not, the feature should be provided as an Extended Telephony service.

All phone-device services are part of supplementary telephony.

Extended Telephony Services

A service provider can extend the Telephony API using device-specific extensions.

The service provider specifies the behavior of all extended services.

There is no requirement that extended services be consistent across devices.

A service provider can extend the standard set of TAPI constants and structures.

The service provider must uniquely identify the extensions by using an *extension ID*. Applications retrieve and use this unique identifier to determine what extensions the service provider supports.

The extensions may apply to several manufacturers.

The service provider Special functions and messages such as **lineDevSpecific** and **phoneDevSpecific** are provided in the API to allow an application to directly communicate with a

They are used to allow extending the set of functions (in contrast to enumerations, bit flags, and data structure fields) supported by the service provider.

The parameters for each function are also defined by the service provider.

An ID is assigned to a set of extensions (before distribution), not to each individual instance of an implementation of those extensions.

To ensure that the extension ID for a service provider is unique

The EXTIDGEN utility generates unique extension IDs for service providers. It uses an Ethernet-adapter address, a random number, and the time of day to generate an identifier, so it is extremely unlikely that the resulting extension ID will conflict with any other service provider. This means there is no need for vendors to register extension IDs.

A range of values is reserved to accommodate future extensions to the basic and supplementary function set.

There is a range reserved for future definition of structures and types within the Windows Telephony specification

separate range reserved for provider-specific extensions.

For information about whether a given structure or set of constants is extensibile, see the individual structure and constant descriptions.

The following structures and constants are extensible:

Addresses

An address is the telephone number, complete with national or international codes, of a telephone, fax machine, or other device that can receive calls. Addresses can be dialed by a human or stored in an electronic directory for retrieval and use (dialing) by a telephony application.

The service provider controls the local addresses of a line.

A line can have a local address. The user or an application sets this address when installing the service provider that suports the line.

The service provider has a configuration. The service provider gives users access to the configuration settings by providing a **TSPI_lineConfigDialog** function.

In the simplest case, each line has one address. However, any line can have more than one address.

In POTS, multiple addresses work only with systems that support distinctive ringing or are connected to a DID trunk.

DID (direct inward dialing) is an extra-fee service provided by the phone company. With DID in a multi-user voice mail system, the dialed number is signaled to the system on the DID trunk before the call rings. This allows the system to play the called party's pre-stored announcement message and to store any incoming messages in the correct voice mail box.

On a residential line with distinctive ringing service different ringing patterns correspond to multiple numbers assigned to the same line.

ISDN allows simultaneous multiple addresses by providing multiple channels, each having a unique address.

On an ISDN network, call offering is the process of sending a call-setup message from the switch. This occurs *before* ringing, so the call can be redirected before it is answered. The **lineAccept** function means "start ringing" for ISDN. For POTS, it means that some application has accepted responsibility for the call and has presented it to the user.

Call Control

The service provider provides useful telephony functions.

TAPI uses a first-party call-control model on the logically terminated line as well as control of the associated phone device, if any.

Applications access Telephony services using a *first-party* call control model.

Service providers must allow applications to control telephone calls as they were an endpoint (the initiator or the recipient) of the call.

The application makes calls, be notified about inbound calls, answer inbound calls, invoke switch features such as hold, transfer, conference, pickup, and park, and can detect and generate DTMF tones for signaling remote equipment. An application can also use TAPI functions to monitor call-related activities occurring in the system.

In contrast, *third-party* call control means that the controlling application does not act as an endpoint of the call. A third-party call-control model allows an application to establish or answer a call between any two parties—the application does not act as either of these parties.

A service provider can treat a set of stations on the switch as a single line device to which multiple phone numbers are assigned. Each phone number on the line device maps to one of the stations on the switch. The application can answer calls or make calls, selecting any one of the addresses on the line device as the origination number. Although the application appears to be the originating party, a call is actually established between the station whose address was selected by its originating number and the other party. However, this implementation is a type of third-party call control and is not a design goal of TAPI, which emphasizes first-party call control applications.

Each individual station on the switch is identified by its primary phone number (or other means). A possible implementation of the SPI's line and phone behavior by a service provider is to model the collection of all stations on the switch as a single line device that has multiple phone numbers assigned to it; each phone number on the line device in reality maps to one of the stations on the switch.

TAPI can make and answer calls selecting any one of the addresses on the line device as the origination number.

Although TAPI.DLL appears to be the originating party, in reality a call will be established between the station whose address was selected by its originating number and the other party.

A distributed service provider supports calls shared by multiple computers in a local area network by communicating with service providers on the other computers.

Media Access

The *media mode* is the form in which data is transmitted on a line. The four main types of media mode are voice, speech, facsimile, and data. The *media stream* is the actual stream of information that travels on the line.

Calls can be established independently of the media mode.

The TSPI line and phone device classes only provide *control* operations for these devices.

Access to media streams is not provided by the Windows Telephony SPI.

Applications must use other Windows APIs to provide this access or otherwise manage these media streams such as the waveform API or the higher-level MCI interface.

A TSPI function identifies the appropriate media device, such as a wave device, to open and use to generate and access the media stream.

The service provider determines the relationship between the phone or line device and the media device.

For line devices, the service provider receives a request to establish a connection to another station. Once a call is established, the service provider receives a request for the media device associated with the call device. After information has been transmitted or received on the media device, the service provider receives a request to shut down the call.

To provide both telephony API functions and media stream access to a phone or to a call on a line device,

The service provider must provide the appropriate media stream functions or the underlying support for them.

The service provide must provide the TSPI function that specifies the relationship between the phone or line devices and the media device.

The physical device supports line, phone, call and wave device classes.

Each telephony device class has a corresponding media device class.

A TSPI function retrieves device-class-specific device identification for a given line, address, call, or phone device.

The service provider reports media stream type changes when requested. This process is sometimes referred to as *call classification*.

The mechanism used to determine the type of a media stream (such as voice, fax, data modem) is service-provider specific.

A service provider may filter the media stream for energy or tones that characterize the media type. Alternatively, the service provider may determine the type by messages exchanged over the network, by use of distinctive ringing, or by knowledge about the caller or called ID.

Media Control

Some TSPI functions provide limited support for control of a media stream. A service provider typically implements these functions to prevent data loss in configurations in which the information of a call is passed between client and server computers.

If a service provider supports media control, it must accept requests to carry out actions on the media stream when given events occur, such as on receiving a specific tone or DTMF digit in the media stream. For example, the service provider might receive a request to suspend the media stream when it detects a # DTMF digit and resume the media stream when it detects a * DTMF digit.

Media control is optional.

Service Provider

A service provider is a dynamic link library that exports TSPI functions.

The variations to the standard DLL interface model that a service provider must observe are as follows:

Use the filename extension TSP instead of DLL. Although, not a strict requirement, using this extension lets users change the service configuration by using the XXX utility.

Service providers that support interactive configuration features and wish to let them be accessed with this tool must have the “.TSP” file extension.

Initialization

The **TSPI_providerInit** and **TSPI_providerShutdown** functions are for initialization and shutdown.

Because TSPI defines initialization and shutdown functions, the service provider does not need to provide an initialization routine or WEP function as used in other dynamic link libraries. The initialization and shutdown functions may be called multiple times while the service provider is active (in memory).

Because the service configuration may change while a service provider is active (in memory), the service provider must reinitialize all configuration-related information whenver the **TSPI_providerInit** function is called. All stored information from previous invocations may have become invalid.

Entry Points

TAPI links to and calls the TSPI functions using the standard dynamic link functions. The service provider must export the TSPI functions using the following ordinal numbers:

Callback Functions

The service provider reports events by calling TAPI-supplied callback functions.

Structures

A service provider uses structures to receive and return the data associated with a telephony request. In many cases, the members in a structure vary in length or number depending on the context of the specific request.

Rather than use pointers to these variable members,

The structures do not contain pointers.

```
DWORD  dwTotalSize;
DWORD  dwNeededSize;
DWORD  dwUsedSize;
    // fixed size fields
DWORD  dwField1Size;
DWORD  dwField1Offset;
    // fixed size fields
DWORD  dwField2Size;
DWORD  dwField2Offset;
    // common extensions
    // var sized field1
    // var sized field2
```

A structure consists of a fixed set of members followed by zero or more variable members.

The first three fixed members specify the size of the structure. The **dwTotalSize** member, set by the application and verified by TAPI, specifies the total size in bytes of memory allocated for the structure. The **dwNeededSize** member specifies the total size in bytes needed for the data to be filled into the structure. The **dwUsedSize** member indicates the total size in bytes of the data that was actually filled in.

Each variable member has a pair of corresponding fixed members that specify the size and the offset, in bytes, from the beginning of the structure to the variable member.

Before passing a structure to the service provider, TAPI ensures that the memory for the structure is writable and that the value in **dwTotalSize** value covers at least the fixed part of the structure. TAPI sets the **dwUsedSize** and the **dwNeededSize** members to the size of the fixed part.

The service provider must:

If the service provider is responsible for filling in a variable member, TAPI sets the corresponding size and offset members to zero. If the service provider fills in the variable member, it must set the corresponding size and offset members to appropriate values. The service provider must *not* truncate a variable member to make it fit in the available space.

The service provider must

Set the fixed and variable members for which it is responsible.

Update the **dwUsedSize** member if it sets variable members. This must be the size of all fixed members plus the size of all variable members the service provider was able to copy to the available space.

Update the **dwNeededSize** member if has any variable members. This must be the size of all fixed members plus the size of all variable members, not just those copied to the available space.

The service provider must start variable members immediately after the fixed members of the structure. It can place the variable members in any order, but make sure the members are contiguous.

If the service provide detects an error, it must return the appropriate error return; no members need to be filled in. Insufficient space for variable members is not an error. The service provider simply sets **dwUsedSize** and **dwNeededSize** as appropriate and returns success. Unless an error occurs, the

service provider must set all fixed members for which it is responsible.

TAPI sets the fixed and variable members that it is responsible for *after* the service provider.

Life Cycle

TAPI loads the service provider .... .

TAPI frees the service provider ... .

TAPI initializes the service provider ...

TAPI shuts down the service provider ...

Service provider can remain loaded even though

If configuration information in TELEPHON.INI changes, the service provider must read the file and determine the new configuration.

A session is the time during which a particular configuration remains valid and telephony operations are carried out.

A service provider may support many sessions between the time it is first loaded and finally freed.

For each session, TAPI negotiates the version of the interface, starts the session, carries out operations, and eventually shuts down the session.

The service provider must not retain information from one session to the next.

Version negotiation consists of TAPI sending the range of versions of the interface that it can support and the service provider responding with the version or versions in that range that it can support.

Because TAPI negotiates the version, you can upgrade a service provider to new versions of the interface without requiring that TAPI be upgraded too. Similarly, TAPI can be upgraded yet still use your older service provider.

Once the interface version is known, TAPI calls the **TSPI_providerInit** function set all operational parameters. The service provider is guaranteed that all of the configuration information in TELEPHON.INI is stable when the **TSPI_providerInit** function is called. Most service providers read all configuration information at that time.

Normal operations can proceed in any order once the service provider is initialized.

TAPI negotiates device-specific on a per-device basis. TAPI and the service provider commit to a version when a device is opened.

The service provider can receives requests to select and cancel extension versions. While an extension version is selected, the device operates strictly according to that device-specific extension version.

Negotiation of a device-specific extension version can happen multiple times, both before and after the phone is opened. TAPI passes a version range, the service provider chooses and returns a value from this range. Normally, the service provider has device-specific extensions disabled.

This commits both TAPI.DLL and the service provider to operating at that extension version level until the selection is canceled. During the time that a device-specific extension is in effect, an attempt to negotiate an extension version level should allow only the version level that is currently in effect. After the device-specific extension is canceled, a different version can be negotiated and selected.

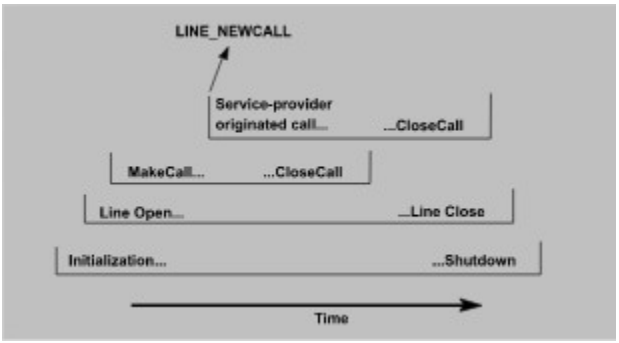
Phone operations within the Open/Close pair are shown in the preceding illustration. Some of these operations are synchronous, others are asynchronous. If an operation completes asynchronously, another operation may be requested before the first reports completion. Thus operations can overlap in any way. The service provider must eventually report completion for any asynchronous operation requested. Closing the phone forces outstanding asynchronous operations to complete (possibly with a “failure” indication).

The life cycle for line devices is similar to the life cycle for phones, except that lines have their own negotiation, initialization, open, and close procedures. Operations on open lines are bracketed by their own Open/Close pair. This pair is in turn bracketed between the same Initialization and Shutdown pair as the phone Open/Close.

The life cycles of calls are bracketed strictly between the Open/Close of the line that contains them. Lifetimes of calls can begin in several ways:

- Requested from TAPI.DLL through functions such as **lineMakeCall**, **lineSetupTransfer**, or **lineSetupConference**.
- Spontaneously originated in the service provider as new incoming calls, user-initiated calls on an attached telephone handset, or calls generated as a side-effect of other operations such as putting an existing call on hold.

The lifetimes of distinct calls within the same line can overlap each other in any way. All calls end their lifetime at the time the TSPI function **TSPI_lineCloseCall** is invoked. An example of several call life cycles is as follows:

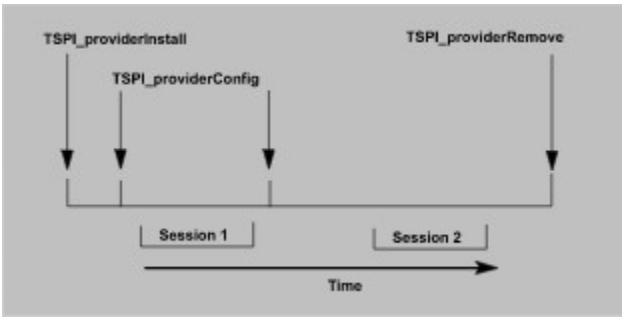


The Life Cycle of Calls

The life cycle of several calls is illustrated above. A call can originate in TAPI.DLL as shown with the MakeCall ... CloseCall pair. A call can also originate in the service provider. The service provider announces this with a **LINE_NEWCALL** message to a TAPI.DLL-supplied callback procedure. In such a case, the TAPI.DLL returns its identifier for the call, which is included in subsequent callbacks reporting events occurring on the call. In the case of calls whose lifetime originates in TAPI.DLL, this identifier is included in the TSPI operation that creates the call.

All of the operations that start the lifetime of a call result in TAPI.DLL and service provider exchanging identifiers for the new call. In the case of TAPI.DLL originated calls, TAPI.DLL passes its identifier and receives the service provider's identifier as a return parameter. In the case where the call originates in the service provider, the service provider passes its identifier to TAPI.DLL and receives TAPI.DLL's identifier as a return parameter.

So far, this section focusses in on more and more fine-grained life-cycle details. The following takes a very high-level view of service provider Installation, Configuration, and Removal; life-cycle sequences that span many sessions. The typical life-cycle for these operations can be shown with the following time line:



Installation, Configuration, and Removal

The typical Installation and Removal life cycle is shown, spanning multiple sessions. Calls to the "Install" and "Remove" procedures are strictly paired, non-overlapping. Calls to the "Config" procedure can happen multiple times within this pair. One is typically done by the service provider as an internal side-effect of "Install" to create line and phone entries. "Config" may be called at other times to alter an existing setup. The "Install" must be done before any other TSPI function is permitted. In the ideal scenario, all sessions are strictly nested within the "Install" and "Remove" pair.

The functions **TSPI_providerInstall, TSPI_providerConfig,** and **TSPI_providerRemove** interact with

the service provider itself rather than with any specific device. They affect static configuration information that survives across multiple sessions and must be present for any other operation to proceed. Thus all other operations are nested between the invocation of **TSPI_providerInstall** and the completion of its matching **TSPI_providerRemove**. These two operations typically happen very far apart, most likely in a different load of the service provider (a different **LibMain/WEP** pair) or a different boot of the machine. Calling the “Config” function externally is optional since the “Install” procedure is required to include the “Config” behavior in addition to its own. The usual reason for calling it externally is to modify an existing configuration.

There is a subtlety embedded in the concept of “the completion of a **TSPI_providerRemove** operation.” It is desirable to allow the user to run the telephony control panel applet supplied with the telephony service to alter service provider configurations even when telephony operations (sessions) are in progress. Consequently, both the **TSPI_providerConfig** and **TSPI_providerRemove** specifications allow for invocation while there is an outstanding session. However, any changes to the configuration are required to be delayed until telephony operations are shut down and restarted. Thus, strictly speaking, the completion of any **TSPI_providerConfig** or **TSPI_providerRemove** operation happens outside any session. The nesting of actions is as shown in the figure even though the procedures calls may appear in a slightly different order. It is permitted for a service provider to simply disallow “Config” or “Remove” while operations are in progress, although the user should be notified with a dialog box. A more user-friendly implementation that allows at least a subset of operations is preferred.

These Install, Config, and Remove operations all have the side-effect of signalling any running telephony applications, eventually resulting in them shutting down their use of the telephony service. Together, this ends all outstanding sessions for service providers. This signifies to service providers that they must read the new TELEPHON.INI configuration when new sessions begin. Any changes that were pending due to a “Config” or “Remove” while operations were in progress then take effect.

The Synchronous/Asynchronous Model

Operations complete either synchronously or asynchronously. These two models differ as follows:

- Asynchronous operations have a formal parameter named *dwRequestID* of the defined type **DRV_REQUESTID** as their first formal parameter. Such an operation performs part of its processing in the function call and the remainder of its processing in an independent execution thread after the function call has returned. When the function returns, it returns either a negative error result or the (positive) *dwRequestID* that was passed in the function call. The negative error result indicates that the operation is complete (and failed). The positive *dwRequestID* returned indicates that the operation continues in the independent thread. In such a case, the service provider eventually calls the **ASYNC_COMPLETION** procedure, passing the *dwRequestID* and a result code to indicate the outcome of the operation. The result code is zero for success or one of the same set of (negative) error results. In any case, the service provider must never return zero or any positive value other than *dwRequestID* from a function designated as Asynchronous because doing so may produce unexpected results.
- Synchronous operations do not have a formal parameter named *dwRequestID* as their first formal parameter. Such an operation performs all of its processing the caller's execution thread, returning the final result when it returns. The result is zero for success or a negative error code for failure. The service provider does not call **ASYNC_COMPLETION** for synchronous operations.

The Meaning of SUCCESS

When an operation returns a SUCCESS indication (either synchronously upon function return for synchronous operations, or asynchronously with a call to the **ASYNC_COMPLETION** callback procedure for asynchronous operations) then the following are assumed to be true:

- The function has successfully progressed up to a service-provider-defined point. This point is defined by the service provider on a function-by-function basis. After reaching that point, either the operation is completely done, or it is in a state such that subsequent independent state messages will inform the application about progress.
- Functions that return information (such as **TSPI_linePark** or **TSPI_lineDevSpecific**) return SUCCESS only when the requested information is available to the caller. Functions that return handles (to opened lines, opened phones, or calls) return the handles immediately on function return. The handles must be treated by both the service provider and the caller as “not yet valid” until the eventual SUCCESS indication. The service provider is responsible for preventing any callback messages to the **LINEEVENT** or **PHONEEVENT** procedure about that opened line, opened phone, or call until after the SUCCESS indication. If the eventual result is an error, any handle returned immediately becomes invalid without any further action.
- Functions that enable certain permanent conditions (like **TSPI_lineMonitorDigits**) return SUCCESS only after the condition is enabled, not when the condition is removed again (for example, not when all digit monitoring has completed).
- Call control functions (such as **TSPI_lineHold** and **TSPI_lineSetupTransfer**, but not **TSPI_lineMakeCall**) return SUCCESS when the operation is completed. In telephony environments that do not provide positive acknowledgment of these functions, the service provider is forced to make a best-effort decision as to the success or failure of the function.
- A service provider’s implementation of **TSPI_lineMakeCall** should return SUCCESS no later than when the call enters the *proceeding* call state. Ideally, the provider should indicate SUCCESS as soon as possible, such as when the call enters the *dialtone* call state (if applicable). Once SUCCESS has been returned to the application, **LINE_CALLSTATE** messages inform the caller about the progress of the call. Service providers that delay returning the **TSPI_lineMakeCall** SUCCESS indication, say, until after dialing is complete must be aware that this places that provider in a disadvantage in that the usability at the application level may be severely limited. For example, it would not be possible for a user to cancel the call setup request in progress until after dialing is complete and all digits have been sent to the switch.

Line Devices

About Line Devices

Line Devices

About Line Devices

Introduction

A line device is a physical channel composed of a telephone line and the hardware to control it such as a fax board, a modem, or an ISDN card. The simplest kind of line device corresponds to exactly one telephone line. In general, Telephony allows a line to contain a pool of multiple homogeneous channels that can be used to establish calls. A service provider implementing a line as a pool of channels allocates those channels out of a set of physical lines it controls, or multiplexes multiple channels onto a single physical line. Although the term “line” often connotes something with two endpoints, within TAPI the term refers just to one endpoint of a communication channel or point of entry into a telephone switching network.

Depending on the environment, a service provider may restrict calls to one per line as in POTS, may allow switching between several calls on a single line while others go on hold, or it may allow multiple calls to be active concurrently.

Lines are used as entry points for Telephony applications

Although the four lines in the preceeding illustration employ different hardware and are used for different functions, their respective service providers abstract them to the same device type, and they are controlled through the same TSPI functions. In the case of the voice-oriented Line 4, its service provider may allow the line to be directly connected to a telephone handset for incoming or outgoing calls. The service provider may also allow software access to the phone by making it available as an instance of the phone-device class, which is described in later chapters.

Although the four lines in the preceding illustration are composed of different hardware and are used for different functions, they are abstracted to the same device type and are governed by the same rules. Note that the telephones represent not phone devices but line devices used for voice calls. When using such a line device for incoming or outgoing calls, the application might also want to open and control an instance of the phone-device class, which is described in detail in later chapters.

TAPI requires that every TAPI-capable line device support all of Basic Telephony. If an application needs to use capabilities beyond those of Basic Telephony (namely Supplementary or Extended Telephony), it must first determine the line device’s capabilities, which can vary according to network configuration (client versus client-server), hardware, service-provider software, and the telephone network.

The function **TSPI_lineGetDevCaps** queries a specified line device to determine its telephony capabilities. The returned information is valid for all addresses on the line device. The *dwExtVersion* parameter of **TSPI_lineGetDevCaps** indicates the version number of the Extension information requested. If it is zero, no Extension information is requested. If it is non-zero, it holds a value that has already been negotiated for this device with the function **TSPI_lineNegotiateExtVersion**. The service provider should fill in device- and vendor-specific extended information according to the extension version specified.

The service provider should also fill in all the fields of the **LINDEVCAPS** data structure, except for **dwTotalSize**, which is filled in by TAPI.DLL. The service provider must not overwrite the **dwTotalSize** field.

Relationships Between Lines, Channels, and Addresses

Introduction: Lines, Channels, and Addresses

In POTS, exactly one channel exists on a line, and this is used exclusively for voice. With ISDN, at least three (and as many as 31) channels can exist on a line concurrently. The simplest form of Windows Telephony functionality involves POTS applications that handle a single line using its single channel. In POTS, an application that wants to transmit data would communicate over one line, and a voice application would communicate over another line—both of these applications could use the same line, but if so, not at the same time.

In the simplest case, one line has exactly one address (telephone number). In cases where lines carry two or more channels, each channel usually has its own address, which means that a line has as many addresses as it has channels. To make addresses easier to manipulate, TAPI assigns Address IDs to them, in a contiguous range of integers starting with 0. This lets the software interfaces refer to addresses independently of the address syntax on any particular telephone network.

Multiple Addresses on a Single Channel

Some installations support the assignment of more than one address to a single channel. On POTS lines, multiple addresses are made possible by various systems, such as “distinctive ringing” or “DID” (direct inward dialing), which are extra-fee services provided by the telephone company.

Many large corporations use DID for incoming calls. Before a call is connected, its destination extension number is signaled to the PBX, which causes the extension to ring instead of the operator’s phone. An example of distinctive ringing in a private home would be if the parents used one address, the children another, and a fax machine a third. Because only one line connects the house to the telephone network, all phones ring when a call appears, but the ring pattern is different depending on the number dialed by the calling party. With distinctive ringing, the people know who the incoming call is meant for, and the fax machine answers its calls by recognizing its own ringing style.

In ISDN, the various B channels do not normally have separate addresses. Because these B channels are usually on the same address, it is the switch (and not the application or a person who has dialed the number) that assigns calls to these channels.

ISDN Subaddresses

Subaddressing is a capability provided on ISDN lines that allow more information than just a single telephone number to be used when dialing. This additional information may specify an individual telephone extension to ring or, in a computing environment, a particular application to be alerted. Other parameters that can be passed can describe the required aspects of a requested modem connection, such as rate and timing.

Address Configurations, Functions, and Messages

Each line device is assigned one or more *addresses*. An address corresponds to a telephone directory number, and it is actually assigned twice: First, by the telephone company at the switch, and second, by the user while configuring the local system. If a telephone number is changed at the switch, the user will normally need to assign the new number at the local system, although some systems may be sophisticated enough to perform the reassignment without human interaction.

After addresses have been assigned to lines, the service provider assigns *address IDs* to addresses. An address ID is a number between 0 and the number of addresses on the line minus one. Because each address depends on its line to exist, the address's ID is meaningful only in the context of the associated line device. For this reason, an address name consists of not only the address ID, but also an identifier of the line. It serves as a kind of shorthand, an easy way for programmers to identify addresses.

More than one address can be assigned to a line device. (The number of assigned addresses is returned by the function **TSPI_lineGetDevCaps**.) These address-to-line assignments (as modeled by the Telephony SPI) are static in that they change only when reconfigured at the switch or network, which can configure them in any of several ways. The relationship between an address and a line and between the address and other local addresses is known as the address's *configuration*.

Usually, one of the addresses assigned to a line is that line's *primary address*. A primary address is the one address on a line that can uniquely identify the line device. This primary address may also be assigned to other lines as a non-primary address.

Individual addresses are assigned *address ID*s, which are numbers in the range zero to the number of addresses on the line minus one. An address ID is meaningful only in the context of a given line device, so an address is named as the tuple consisting of the line and an address ID. As described later, a line can be named by its device ID or, once it has been Opened, its handle.

The network or switch can configure address-to-line assignments in several ways. Usually, one of the addresses assigned to a line is that line's primary address. A line's primary address is able to uniquely identify the line device, although the address may also be assigned to other lines as a non-primary address. Address configurations recognized by the Telephony SPI include:

1. **Private**. The address is assigned to one line device only. An inbound call for this address is only offered at one line device.
2. **Bridged**. A bridged address is a single address assigned to more than one line device. (Different switch vendors have different names for address bridging, such as "multiple appearance directory number" (MADN), "bridged appearance," or "shared appearance.") Depending on the switch vendor, different terminology may be used such as multiple appearance directory number (MADN), bridged appearance, or shared appearance. An incoming call on a bridged address is offered on all lines associated with the address. The network of lines connected together is known as the "bridge." Different variations of bridged behavior are possible:
   - *Bridged-Exclusive*. Connecting one of the bridged lines to a remote party causes the address to appear "in use" to all other parties that are part of the bridge.
   - *Bridged-New*. Connecting one of the bridged lines to a remote party does not preclude the other lines from using the bridged address to answer or make calls. However, a new call appearance is allocated to another of the connected lines.
   - *Bridged-Shared*. If one line is connected to a remote party, other bridged lines that use the address automatically enter into a multi-party conference call on the existing call.
3. **Monitored**. The line indicates the busy or idle status of the address, but the line cannot use the address for answering or making calls.

Address-Related Functions and Messages

As is the case with line devices, the addresses assigned to a line device may have different capabilities. Switching features and capabilities (including authorization) may be different for different addresses. When an application calls **lineGetAddressCaps**, TAPI.DLL calls **TSPI_lineGetAddressCaps** to determine the telephony capabilities of each address. The service provider must return the telephony capabilities of an address as a data structure of type **LINEADDRESSCAPS**. In a similar way, the service provider implements **TSPI_lineGetDevCaps** to provide an application with the number of addresses assigned to the line, as well as other information.

The SPI's device query capability, and status and event reporting mechanisms provide TAPI.DLL with the information required to manage the different bridged address arrangements. For example, TAPI.DLL can determine if a call has been answered by a bridged station by tracking the status changes and call state event changes on the address (see later sections).

An address on a line device is normally identified through its address ID. When making calls, the SPI allows alternate address selections for the originating address, such as its address in dialable format or even service-provider-specific naming mechanisms (for example, by using SPI extensions based on switch-assigned station IDs). The function **TSPI_lineGetAddressID** maps the alternate address format back to the address ID used by other functions in the SPI.

TAPI.DLL is notified about changes in the status of an address in the **LINE_ADDRESSSTATE** callback message. TAPI.DLL can control the address status items for which it wants to be notified with the function **TSPI_lineSetStatusMessages**. It can determine the complete status of an address device by calling **TSPI_lineGetAddressStatus**.

Basic Line Services

Introduction: Basic Line Services

Line Initialization

As part of the line device abstraction defined by TSPI, TAPI.DLL and the service provider must first perform some basic initialization. Its first step is interface version negotiation. TAPI.DLL performs this by calling the **TSPI_lineNegotiateTSPIVersion** function. This function is usually used to negotiate on behalf of an individual line device; different line devices within the same service provider may operate under different interface versions. TAPI.DLL passes a special reserved device ID value, **INITIALIZE_NEGOTIATION,** to indicate that it is negotiating an overall interface version for initialization functions that affect the entire service provider.

The result of this negotiation is passed to subsequent procedures until a line device is opened. At that time, the line device becomes committed to a particular interface version. This interface version is implicit until the line is closed, and does not need to be passed to subsequent functions that operate on an opened line or calls on the line.

Following overall interface version negotiation, TAPI.DLL calls the function **TSPI_providerInit**. This function initializes the service provider, also giving it parameters required for subsequent operation. These parameters include the following:

dwPermanentProviderID specifies the permanent ID, unique within the service providers on this system, of the service provider being initialized.

dwLineDeviceIDBase specifies the lowest device ID for the line devices supported by this service provider. Devices of the Telephony line device class are identified by integers starting at zero. This range of identifiers is contiguous across the full range of line devices. Since there may be multiple service providers managing line devices in a single system, each service provider obtains a contiguous portion of the total range. This parameter tells the service provider the lowest value in its portion of the range. The service provider, rather than TAPI.DLL, has the responsibility for mapping this variable range to its own internal device identifiers. This gives the service-provider vendor sufficient flexibility to use device IDs in device-specific extensions if it so desires. Since the service provider “knows” what device IDs appear in the TAPI-defined parameters and data structures, it can use consistent device IDs in device-specific extension parameters and data structures.

dwPhoneDeviceIDBase specifies the lowest device ID for the phone devices supported by this service provider.

dwNumLines specifies how many line devices this service provider supports.

dwNumPhones specifies how many phone devices this service provider supports.

lpfnCompletionProc specifies the procedure the service provider calls to report completion of all asynchronously operating procedures on line and phone devices.

Following **TSPI_providerInit**, normal operations such as opening lines can be performed.

Line Configurations

A line device can represent a pool of homogeneous resources (channels) that are used to establish calls. In a client computer, the Telephony SPI typically provides access to one or more line devices.

All line devices support at least the Basic telephony services. If TAPI.DLL's client application is interested in using Supplementary and/or Extended telephony services, the application must first determine the exact capabilities of the line device. Telephony capabilities vary with configuration (such as client versus client/server), hardware, service-provider software, and the telephone network. Applications should not assume what telephony capabilities are available beyond just basic telephony services. TAPI.DLL determines the line's device capabilities on behalf of an application using **TSPI_lineGetDevCaps**, which returns the supplementary and extended capabilities of a given line device as a data structure of type **LINEDEVCAPS**.

Here are some examples of how a service provider might model various configurations:

Example 1 A single POTS line in the computer-centric or phone-centric connection models. This would most straightforwardly be modeled as a single line device with one channel.

Example 2 A single BRI-ISDN line in the computer-centric or phone-centric connection models. The service provider has a number of options as to how it may want to model this; for example:

- Model the BRI line as a single line device with a pool of two channels allowing both channels to be combined for establishing 128 kbps calls.
- Model each B-channel as a separate line device and disallow both channels to be combined into a single 128 kbps channel.
- Model the BRI connection as two separate line devices, each drawing up to 2 channels from a shared pool of 2 B-channels.
- Model as three line devices, one for each B-channel, and one for the combination. Clearly in the latter two models, resources may be assigned to different line devices at different times.

Example 3 In client/server systems, a pool of telephone ports attached to a server may be shared among multiple client computers over a local area network. The pool may be administered to assign a maximum number of line devices (quota) to each client workstation. It is not unusual for the sum of all quotas to exceed the total number of lines. Furthermore, a given client with quota equal to 2 may be satisfied by using ports 1 and 2 at one time and ports 7 and 11 at a later time. The service provider for the pool may model this arrangement by giving each client workstation access to two line devices.

Suppose that the configuration of service providers for a given computer is such that this particular service provider gets a DeviceIDBase of 3. This implies that the (fixed) device IDs for this client are 3 and 4. If the application later calls the TAPI function **lineGetDevCaps** for device 3 and again for device 4, it should be able to assume that the device capabilities for each of these devices are constant, since that is the Windows device model. As long as the given computer's service-provider configuration remains constant, the device IDs that appear at the TSPI level remain constant, even if the server changes port allocations. For server-based devices that are pooled as described in the example above, this holds only for line devices that have identical device capabilities. Of course, if the given computer's service-provider configuration is changed such that this service provider gets a different DeviceIDBase, device IDs change correspondingly.

Example 4 Switch-to-host link.

- To provide personal telephony to each desktop, the service provider could model the PBX line paired with the computer as a single line device with one channel. Each client computer would have one line device available.
- Each third-party station could be modeled as a separate line device. This enables the application to control calls on other stations. This solution requires that the application open each line it wants to manipulate or monitor, which may be fine if only a small number of lines is of interest, but may generate a lot of overhead if a large number of lines is involved.
- The set of all third-party stations could be modeled as a single line device with one address (phone number) assigned to it per station. Only a single device is to be opened, providing monitoring and control of all addresses on the line (all stations). To originate a call on any of these stations, the application only needs to specify the station's address to the operation that makes the call. No extra open operations are required. This modeling does imply that all stations have the same line device

capabilities, although their address capabilities could be different.

Note that all of these examples are simply different ways that a service provider may choose to implement the mapping of line device behavior onto some physical realization.

TAPI.DLL is notified about changes in the status of a line device with the **LINE_LINEDEVSTATE** callback message. TAPI.DLL can control line status items for which it wants to be notified with the function **TSPI_lineSetStatusMessages**. It can determine the complete status of a line device by calling **TSPI_lineGetLineDevStatus**.

Opening and Closing Line Devices

After performing basic initialization and possibly having retrieved device capabilities, TAPI.DLL must open the line device before it can access telephony functions on that line. When a line device has been opened successfully, TAPI.DLL is returned a handle representing the open line. The function **TSPI_lineOpen** opens a specified line device for providing subsequent monitoring and/or control of the line. **TSPI_lineClose** closes a specified line device.

At the API level, a line device can be specified to **lineOpen** in one of two ways:

- A specific line device is selected by means of its line device ID. The **lineOpen** request opens the specified line device, possibly resulting in a **TSPI_lineOpen** call to the service provider (that is, if the line was not already open). Applications interested in handling inbound calls typically use specific line devices.
- The application can specify that it wants to use any line device that has certain properties. The LINEMAPPER indication is used instead of a specific line device ID. The properties are specified by means of the call parameters that the application intends to use. TAPI.DLL satisfies this Open request with any available line device that supports the specified call parameters, but this may fail. If successful, the caller can determine the line device ID by calling **lineGetID** on the handle to the open line device returned by **lineOpen**.

To support LINEMAPPER selection of lines, the service provider must implement the function **TSPI_lineConditionalMediaDetection**. This function does two things: (1) it tests a line's ability to support the requested call parameters and media mode, and (2) if successful, it commits the service provider to monitoring for the new media mode. TAPI.DLL uses this function for different line devices to hunt for a line that meets the application's Open requirements.

Version Compatibility

Over time, different versions of TAPI.DLL, applications, and service providers for a line or phone may be produced. These new versions may make new definitions, such as for new features, new fields in data structures, and new bit fields. Version numbers are therefore necessary to indicate how to interpret various data structures.

Version Negotiation

To allow optimal interoperability of different versions of applications, versions of TAPI.DLL itself, and versions of service providers by different vendors, the TSPI provides a simple version negotiation mechanism for applications. There are two different versions that need to be agreed on by TAPI.DLL and the service provider for each line device. The first is the version number for the Basic and Supplementary Telephony SPI and is referred to as the *TSPI interface version*. The other is for provider-specific extensions, if any, and is referred to as the *extension version*. The format of the data structures and data types used by the SPI's basic and supplementary features is defined by the SPI version, while the extension version determines the format of data structures defined by the vendor-specific extensions.

These two types of version negotiation are handled by two different procedures: **TSPI_lineNegotiateTSPIVersion** is used to negotiate the TSPI interface version, and **TSPI_lineNegotiateExtVersion** is used to negotiate the extension version. Extension version negotiation can always be skipped if extensions are not desired. An extension version of zero is a special value indicating that no extensions are to be used.

For each type of negotiation, TAPI.DLL passes the service provider the highest and lowest version numbers with which it is compatible. The service provider compares this with its own supported range of version numbers. If these ranges overlap, the service provider must return a value within the overlapping portion of the range as the result of the negotiation. Usually, this should be the highest possible value. If the ranges do not overlap, the two parties are incompatible and the function returns an error.

Results of a negotiation simply indicate that the service provider is "willing" to operate at a particular version number, but do not commit the service provider to doing so. For example, TAPI.DLL may re-negotiate to determine an "ideal" version after having negotiated a "possible" version. The TSPI interface version is only committed when a line is opened using **TSPI_lineOpen** and survives until the device is closed. The extension version is committed when the **TSPI_lineSelectExtVersion** function is called, and survives until the selection is canceled by selecting extension version zero.

Extension version selection can happen many times, including while an extension version is in effect. Since the service provider is committed to the extension version, its range of supported versions narrows to exactly that extension version. For example, consider a service provider that is normally compatible with extension versions 1.0 through 5.5. If version 3.0 is in effect while a caller attempts to negotiate a version within the range 1.0 to 5.5, the negotiation returns 3.0.

Address Formats

An address assigned to a line device is typically selected by its address ID. Each address ID corresponds to a directory number through which the address is known to entities that want to make calls to it. Also, making outbound calls typically requires that a directory number be supplied to identify the called party.

Storing Numbers in Electronic Address Books

Many users choose to dial people, fax machines, bulletin boards, and other entities by selecting their names from an address book. The actual number that is dialed depends on the geographic location of the user and the way the line device to be used is connected. For example, a computer may have access to two lines, one connected to a PBX, the other to the telephone company's central office. When making a call to the same party, different numbers may have to be used. (To dial through the PBX, for example, the PC may need to dial '9' to "get out," or a different prefix may be needed for a call made through the central office.) Or, a user may make calls from a portable computer and want to use a single, static address book even when calling from different locations or telephony environments. TAPI's address translation capabilities let the user inform the computer of the current location and the desired line device for the call. TAPI then handles any dialing differences, requiring no changes to the user's address book.

A related topic is the handling of international *call progress monitoring*, which is the process of listening for audible tones such as dialtone, special information tones, busy signals, and ringback tones to determine the state of a call (its "progress" through the network). Because the cadences and frequencies of call-progress tones vary from country to country, the service provider must know what call progress to follow when making an international outbound call. Therefore, applications specify the destination *country code* when placing outgoing calls.

To deal with directory numbers, the API defines two common formats. One is the *canonical* address format, the other is the *dialable* number format. The canonical address format is intended to be a universally constant directory number; it is the directory number one would like to store in the address book, and never change. The API contains operations (address translation) to take addresses in the canonical address format and convert them to the dialable number format. A side effect of this conversion process is to extract a country code identifying the target country for the call. All addresses that appear at the TSPI have been translated to dialable address format, so only the dialable address format is documented below. Functions involving dialing include the extracted country code. This country code can be used by the service provider to provide call progress determination that is independent of national boundaries.

Dialable Addresses

The dialable address format describes a number that can be dialed on the given line. A dialable address contains part addressing information and is part navigational in nature. A dialable address is an ASCII string with the following structure:

DialableNumber **|** *Subaddress* **^** *Name* **CRLF** ...

where:

DialableNumber digits and modifiers **0-9 A-D * # , ! W w P p T t @ $? ;** delimited by **| ^ CRLF** or the end of the dialable address string.

Within the DialableNumber, note the following definitions:

0-9 A-D * # ASCII characters corresponding to the DTMF and/or pulse digits.

! ASCII Hex (21). Indicates that a hookflash (one-half second on hook followed by one-half second off hook before continuing dialing) is to be inserted in the dial string.

P p ASCII Hex (50) or Hex (70). Indicates that pulse dialing is to be used for the digits following it.

T t ASCII Hex (54) or Hex (74). Indicates that tone (DTMF) dialing is to be used for the digits following it.

, ASCII Hex (27). Indicates that dialing is to be paused. The duration of a pause is device specific and can be retrieved from the line's device capabilities. Multiple commas can be used to provider longer pauses.

W w ASCII Hex (57) or Hex (77). An uppercase or lowercase W indicates that dialing should proceed only after a dial tone has been detected.

@ ASCII Hex (40). Indicates that dialing is to "wait for quiet answer" before dialing the remainder of the dialable address. This means to wait for at least one ringback tone followed by several (usually five or more) seconds of silence.

$ ASCII Hex (24). It indicates that dialing the billing information is to wait for a "billing signal" (such as a credit card prompt tone).

? ASCII Hex (3F). It indicates that the user is to be prompted before continuing with dialing. The provider does not actually do the prompting, but the presence of the "?" forces the provider to reject the string as invalid, alerting the application to the need to break it into pieces and prompt the user in-between.

; ASCII Hex (3B). If placed at the end of a partially specified dialable address string, it indicates that the dialable number information is incomplete and more address information will be provided later. ";" is only allowed in the *DialableNumber* portion of an address.

| ASCII Hex (7C), and is optional. If present, the information following it up to the next **+ | ^ CRLF**, or the end of the dialable address string is treated as subaddress information (as for an ISDN subaddress).

Subaddress A variable sized string containing a subaddress. The string is delimited by the next + | ^ **CRLF** or the end of the address string. When dialing, subaddress information is passed to the remote party. It can be such things as an ISDN subaddress or an email address.

^ ASCII Hex (5E), and is optional. If present, the information following it up to the next **CRLF** or the end of the dialable address string is treated as an ISDN name.

Name A variably sized string treated as name information. *Name* is delimited by **CRLF** or the end of the dialable address string. When dialing, name information is passed to the remote party.

CRLF ASCII Hex (0D) followed by ASCII Hex (0A). If present, this optional character indicates that another dialable number is following this one. It is used to separate multiple canonical addresses as part of a single address string (inverse multiplexing).

Calls

Unlike line devices and addresses, *calls* are dynamic. A call or call appearance represents a connection between two (or more) addresses. The originating address (the caller) is the address from which the call originates, and the destination address (the called) identifies the remote end point or station with which the originator wishes to communicate.

Zero, one, or more calls can exist on a single address at any given time. A familiar example of multiple calls on a single address is *call waiting*: During a conversation with one party, a subscriber with call waiting is alerted that another party is trying to call. The subscriber can flash the phone to answer the second caller (which automatically places the first party on hold), and then toggle between the two parties by flashing. In this example, the subscriber has two calls on one address on the line. Since the person at the telephone handset can be talking to only one remote party at a time, only one call is *active* per line at any point in time. The telephone switch keeps the other calls *on hold*. With a line able to encompass more than one channel, different configurations can allow multiple active calls on one line at a time.

The Telephony SPI identifies a specific call by means of a call handle, and TAPI.DLL and the service provider both assign call handles as required. Since each needs to allocate a data structure to maintain the state information it associates with a call, each has its own handle. At the start of the lifetime of a call, TAPI.DLL and the service provider exchange handles with one another. Then when TAPI.DLL calls a function to operate on a call, it passes the service provider's handle. Typically, the service-provider designer implements its handles as pointers or array indexes. This allows simple and efficient access to the associated data structure given the handle. TAPI.DLL uses similar techniques to find its associated data structure when the service provider sends it a message that includes a call handle.

TAPI.DLL tells the service provider to dispose of a call handle by calling **TSPI_lineCloseCall**.

Async Replies versus Events on New Call Handles

The TSPI includes a number of operations that start the lifetime of a call handle. If the service provider returns a “success” indication for such an operation, it must fill in the opaque handle of type HDRVCALL, which it uses for the new call before it returns. TAPI.DLL never performs operations on the call before the matching **ASYNC_COMPLETION** has been received. If the service provider returns a “failure” indication, the handle automatically becomes invalid (that is, without **TSPI_lineCloseCall**). In effect, TAPI.DLL treats the handle as “not yet valid” until the async completion.

The service provider must also treat the handle as “not yet valid” until after the matching **ASYNC_COMPLETION** has been received. In other words, it must not issue any **LINEEVENT** messages for the new call or include it in call counts in messages or status data structures for the line.

Note that the TAPI level also conforms to these life cycle restrictions; a new call handle is not returned or valid until the matching **LINE_REPLY** message, and no events for the new call, are delivered to the application until after the **LINE_REPLY** message.

The TSPI operations to which this “start lifetime” principle applies are the following:

TSPI_lineCompleteTransfer

TSPI_lineForward

TSPI_lineMakeCall

TSPI_linePickup

TSPI_linePrepareAddToConference

TSPI_lineSetupConference

TSPI_lineSetupTransfer

TSPI_lineUnpark

Making Calls

Before an outbound call can be made, TAPI.DLL must have opened the line device. The standard way to make the call is then for TAPI.DLL to invoke **TSPI_lineMakeCall** specifying the line handle and a dialable destination address. The request first obtains a call appearance on an address on the line, waits for dialtone, and dials the specified address. If successful, TAPI.DLL's handle to the new call is stored by the service provider for subsequent event reports and the service provider's handle to the new call is returned. TAPI.DLL saves the service provider's handle for subsequent requests on the new call.

When making a new call, TAPI.DLL has the option of specifying the address on the line where it wants the call to originate. This can be done by specifying the address ID or by using the corresponding directory number. The latter may be useful in configurations where it is more practical to identify the originating address by its directory number than by its address ID. If special call setup parameters are to be taken into consideration, TAPI.DLL must supply additional information to **TSPI_lineMakeCall**. Call setup parameters are required when requesting such things as a special bearer mode, a call's bandwidth, a call's expected media mode, user-to-user information (for ISDN), securing of the call, blocking the sending of caller ID to the called party, or automatically taking the phone offhook at the originator and/or the called party.

TAPI.DLL can also use **TSPI_lineMakeCall** only to allocate a call appearance (or partially dial) and then explicitly perform (or complete) dialing by using **TSPI_lineDial**. When the number provided is incomplete, dialing the digits may be delayed by placing a ';' (semicolon) at the end of the number. The function **TSPI_lineMakeCall** makes an outbound call and returns a call handle for the new call. **TSPI_lineDial** dials (parts of one or more) dialable addresses. Use this operation in all situations where you need to send address information to the switch on an existing call, such as dialing the address of a party to which to transfer a call.

Once dialing is complete, call progress information is provided to TAPI.DLL in the **LINE_CALLSTATE** callback message. This enables TAPI.DLL to track whether or not the call is reaching the called party.

The dialable number format allows multiple destination addresses to be supplied at once. This may be useful if the service provider offers some form of inverse multiplexing by setting up calls to each of the specified destinations and then managing the information stream as a single high-bandwidth media stream. TAPI.DLL would perceive this as a single call, as it receives only a single call handle representing the aggregate of all the individual phone calls. It is also possible to support inverse multiplexing at the application level. Then the application would establish a series of individual calls and synchronize their media streams.

Call Notification

After TAPI.DLL has opened a line device and has called **TSPI_lineSetDefaultMediaDetection** to specify the desired media modes for inbound calls, it can be notified when a call arrives. The service provider informs TAPI.DLL of a new incoming call in the **LINE_NEWCALL** callback message. This message passes the service provider's handle for the call to TAPI.DLL. TAPI.DLL returns its handle for the call. The service provider follows this with a **LINE_CALLSTATE** callback message. For an unanswered inbound call, the call state is *offering*. TAPI.DLL can then invoke **TSPI_lineGetCallInfo** to find out information about the call. The fact that a call is offered may not necessarily imply that the user is being alerted. A separate **LINE_LINEDEVSTATE** callback is made with a *ringing* indication to provide this information to TAPI.DLL.

Note that **LINE_NEWCALL** may fail due to finite-resource limitations. TAPI.DLL may run out of resources to handle a new call under very high traffic conditions. In such a case, TAPI.DLL returns NULL instead of a valid call handle, indicating that the newly offered call handle was not succesfully received. The service provider may respond by dropping the call or by retrying the **LINE_NEWCALL** after a scheduling delay. In any case, the service provider must check for this NULL return and must never attempt to pass subsequent messages using the invalid call handle.

The **LINE_NEWCALL** message and subsequent **LINE_CALLSTATE** message also notifies TAPI.DLL about the existence and state of outbound calls established as a side-effect of other calls (for example, when an active call is put on hold and replaced by a new call in dialtone state) or manually by the user (for example, on an attached phone device). The call state of such calls reflects the actual state of the call, which will not be offering. By examining the call state, TAPI.DLL can determine whether the call is an inbound call that needs to be answered.

Call information includes (among other things):

- **Bearer mode, rate** This is the bearer mode (voice, data) and data rate (in bps) of the call.
- **Media mode** The current media mode of the call. *Unknown* if this information is unknown.
- **Call origin** Indicates whether the call originated from an internal caller, an external caller, or unknown.
- **Reason for the call** Describes "why" the call is occurring. Possible reasons are: direct call, transferred from another number, busy-forwarded from another number, unconditionally forwarded from another number, the call was picked up from another number, a call completion request, or a callback reminder. The reason for the call is given as *Unknown* if this information is not known.
- **Caller-ID** Identifies the originating party of the call. This can be in a variety of (name or number) formats, determined by what the switch or network provides.
- **Called-ID** Identifies the party originally dialed by the caller.
- **Connected-ID** Identifies the party that is actually connected to. This may be different from the called party if the call was diverted.
- **Redirection-ID** Identifies to the caller the number towards which diversion was invoked.
- **Redirecting-ID** Identifies to the diverted-to user the party from which diversion was invoked.
- **User-to-user information** User-to-user information sent by the remote station (ISDN).

Note that depending on the telephony environment, not all information about a call may be available at the time the call is initially offered. For example, if caller ID is provided by the network between the first and second ring, the caller ID is still unknown when the call is first offered. When it becomes known shortly thereafter, a **LINE_CALLINFO** callback message notifies TAPI.DLL about the change in party-ID information of the call.

If a new call arrives while another call already exists on the line or address, similar notification and call information is supplied following the same mechanism as for any incoming call. If an application does not want any interference for a call from the switch or phone network, it should secure the call. Securing a call can be done at the time the call is made using a parameter of **TSPI_lineMakeCall**, or later, with the function **TSPI_lineSecureCall**, when the call already exists. The call is secured from interference by other events until it is disconnected. Securing a call may be useful, for example, when it is feared that certain network tones (such as call waiting) could disrupt a call's media stream, such as fax.

Once TAPI.DLL has been offered a call, it may answer the call using **TSPI_lineAnswer**. Once a call

has been answered, its state typically transitions to *connected* and information can be exchanged over it.

Resource Limitations

Once TAPI.DLL has opened a line device, it can make outbound calls on the line. Note that a successful Open of a line device does not automatically guarantee that the **TSPI_lineMakeCall** request will succeed. Depending on the environment, a service provider may have limits on the number of concurrent calls on a line. The service provider may return a failure in response to **TSPI_lineMakeCall** if it is unable to allocate line resources necessary for a call appearance for the desired new outbound call. If other calls exists on the line, these calls would have to be on hold and would typically be forced to remain on hold until TAPI.DLL either places the new call on hold or drops the call. For service providers that model lines as a pool of available channels, these restrictions may not apply, since the service provider may have sufficient channel resources to make a new outbound call, even when other channels in the pool are in use.

Whenever TAPI.DLL has opened a line device, it is notified about certain general status and events occurring on the line device or its addresses. These include the line being taken out of service, the line going back in service, the line being under maintenance, or an address becoming in use or going idle. TAPI.DLL will call **TSPI_lineClose** on the line device handle after all client applications have called **lineClose** on the API.

Note that in certain environments, it may be desirable for a line device that is currently open to be forcibly reclaimed (possibly by the use of a small control application) from TAPI.DLL's control. This might be required, for example, if the service provider detects a situation where the line must be taken out of service or a PBX administrator decides to preempt use of a line. If a service provider encounters such a situation, it may issue a **LINE_CLOSE** message to TAPI.DLL for the open line device that was forcibly closed.

Setting a Terminal for Phone Conversations

The user's computer may have access to multiple devices that are selectable by the user and used to conduct interactive voice conversations. First, there is the phone device itself, complete with lamps, buttons, display, ringer, and voice I/O device (handset, speakerphone, headset). The user's computer may also have a separate voice I/O device (such as a headset or a microphone/speaker combination attached to a sound card) for use with phone conversations.

The Telephony SPI enables the user to select where to route the information that the switch sends over the line, address, or call. The switch normally expects this to be one of its phone sets, and sends ring requests, lamp events (for stimulus phones), display data, and voice data as appropriate. The phone in turn sends hookswitch events, button press events (for stimulus phones), and voice data back to the switch. Note that the *line* portion of the Telephony SPI makes lamp events, display events, and ring events available, either as functional return codes to the various line operations in the SPI, or as unsolicited functional call status messages that are sent to the application callback. The service provider implementation is responsible for mapping between the functional SPI level and the underlying stimulus or functional messages used by the telephony network. In functional telephony environments, the SPI functions are mapped to the functional protocol.

Using the **TSPI_lineSetTerminal** function, TAPI.DLL can control the routing of different low-level events exchanged between the switch and the station; it can also decide not to route some of these low-level events to a device. The routing of the different classes of events can be individually controlled. For example, the media stream of a call (such as voice) can be sent to any transducer device if the service provider and hardware are capable of doing so. Ring events from the switch to the phone can be mapped into a visual alert on the computer's screen or they can be routed to a phone device. Lamp events and display events can be ignored or routed to a phone device (which appears to behave as a normal phone set). Finally, button presses at a phone device may or may not be passed on to the line. In any case, this routing of low-level signals from the line does not affect the operation of the line SPI portion which always maps the low-level events to their functional equivalent. Applications may consult a line's device capabilities to determine the terminals it has available.

In other words, the function **TSPI_lineSetTerminal** specifies the terminal device to which the specified line, address events or call media stream events are routed. Separate terminals can be specified for each event class. Event classes include lamps, buttons, display, ringer, hookswitch, and media stream.

Call Drop

To terminate a call or a call attempt, TAPI.DLL can use **TSPI_lineDrop** on the call. This has the effect of hanging up on that call and making it possible to make another call. This function disconnects a call, or abandons a call attempt in progress.

If the remote party disconnects a call, the service provider sends a **LINE_CALLSTATE** message with a call state of *disconnected*. To clear the call, TAPI.DLL must also drop the call by invoking **TSPI_lineDrop** on the call, which causes the call to be transitioned by the service provider to the *idle* state.

A call handle remains valid after a call has been dropped. This enables TAPI.DLL to still call operations such as **TSPI_lineGetCallInfo** to retrieve information about a call (for example, for logging purposes). The call handle is eventually deallocated by **TSPI_lineCloseCall**., which shuts down operations for a call and deallocates (invalidates) the call handle

Call Handle Manipulation

The API provides a number of functions for manipulating call handles, determining the relationship between lines, calls, and address, and so on. Most of this functionality is implemented strictly within TAPI.DLL without reference to the service provider, except for **TSPI_lineGetCallAddressID**. This procedure retrieves the address ID of an existing call within its line. Service providers that model a line as a group of address IDs may choose an unpredictable address for a new inbound or outbound call. This function gives TAPI.DLL sufficient information to implement the **lineGetNewCalls** operation when it is invoked with the **LINECALLSELECT_ADDRESS** option.

Supplementary Line Services

Introduction: Supplementary Line Services

Bearer Mode and Rate

The notion of *bearer mode* corresponds to the quality of service requested from the network for establishing a call. It is important to keep the concept of bearer mode separate from that of media mode. The *media mode* of a call describes the type of information that is exchanged over a specific call of a given bearer mode. As an example, the analog telephone network (PSTN) only provides 3.1 kHz voice grade quality of service—this is its bearer mode. However, a call with this bearer mode can support a variety of different media modes such as voice, fax, or data modem. In other words, media modes require certain bearer modes. The Telephony SPI only manages the bearer modes by passing the bearer mode parameters on to the network. Media modes are fully managed through the appropriate media mode APIs, although some limited support is provided in the Telephony SPI.

The bearer mode of a call is specified when the call is set up, or is provided when the call is offered. With line devices able to represent channel pools, it is possible for a service provider to allow calls to be established with wider bandwidth. The *rate* (or bandwidth) of a call is specified separately from the bearer mode, allowing an application to request arbitrary data rates.

The bearer modes defined in Windows Telephony are:

- **Voice** Regular 3.1 kHz analog voice service. Bit integrity is not assured.
- **Speech** G.711 speech transmission on the call.
- **Multiuse** As defined by ISDN.
- **Data** Unrestricted data transfer. The data rate is specified separately.
- **Alternate speech and data** The alternate transfer of speech and unrestricted data on a call (ISDN).
- **Non call-associated signaling** This provides a clear signaling path from the application to the service provider.

Although support for changing a call's bearer mode or bandwidth is limited in networks today, the SPI provides an operation to request a change in specific call parameters, namely the call's bearer mode and/or data rate. The operation is **TSPI_lineSetCallParams**.

Media Monitoring

When a call is in the *connected* state, information can be transported over the call. A call's *media mode* provides an indication of the type of information (for example, the data type or higher-level protocol) of this media stream. The Telephony SPI allows TAPI.DLL to be provided with a callback notification about changes in a call's media mode. The notification provides an indication of the call's new media mode. Note that the service provider decides how it wants to make this determination. For example, the provider could use signal processing of the media stream to determine media mode, or it could rely on distinctive ringing patterns assigned to different media streams, or on information elements passed in an out-of-band signaling protocol. Independent of how the media mode determination is done, TAPI.DLL is simply informed about media mode changes on a call.

The media modes defined by Windows Telephony include:

- **Unknown** The media mode of the call is not currently known—the call is unclassified.
- **Interactive voice** Voice energy was detected on the call, and the call is handled as an interactive voice call with a person at the application's end.
- **Automated voice** Voice energy was detected on the call, and the call is handled as a voice call but with no person at the application's end, such as with an answering-machine application. When the service provider cannot distinguish between interactive and automated voice on an incoming call, it should treat or report the call as interactive voice.
- **Data modem** A modem session on the call. Note that current modem protocols require the called station to initiate the handshake. For an inbound data modem call, the application can typically make no positive detection. How the service provider makes this determination is its choice. For example, a period of silence just after answering an inbound call may be used as a heuristic to decide that this might be a data modem call.
- **G3 fax** A group 3 fax session on the call.
- **TDD** The call's media stream uses the Telephony Devices for the Deaf protocol.
- **G4 fax** A group 4 fax session on the call.
- **Digital data** A digital data stream of unspecified format.
- **Teletex, Videotex, Telex, Mixed** These correspond to the telematic services of the same names.
- **ADSI** An Analog Display Services Interface session on the call. ADSI enhances voice calls with alphanumeric information downloaded to the phone and the use of soft buttons on the phone.

Media monitoring can be enabled and disabled on a specified call with **TSPI_lineMonitorMedia**. TAPI.DLL specifies which media modes it is interested in monitoring. If enabled, the detection of a media mode causes TAPI.DLL to be notified in the **LINE_MONITORMEDIA** callback message, which notifies TAPI.DLL about a media mode change. The callback provides the call handle on which the media mode change was detected as well as the new media mode.

There is a distinction between the media mode of a call as reported by **TSPI_lineGetCallInfo** and the media mode event reports in **LINE_MONITORMEDIA** messages. A call's media mode is determined exclusively by TAPI.DLL and is not automatically changed by media monitoring events.

Default media mode monitoring for calls is performed for the media modes for which the line device has been set. This allows an inbound call's media mode to be determined early based on what is needed by TAPI.DLL.

Note that the scope of a call's media monitoring of a call is bound by the lifetime of the call. Media monitoring on a call ends as soon the call *disconnects* or goes *idle*.

TAPI.DLL can obtain device IDs for various Windows device classes associated with an opened line by invoking **TSPI_lineGetID**. This function takes a line handle, address, or call handle and a device class description. It returns the device ID for the device of the given device class that is associated with the open line device, address, or call. If the device class is "tapi/line," the device ID of the line device is returned. If the device class is "mci wave," the device ID of an mci waveaudio device (if supported) is returned that allows the manipulation such as recording or playback of audio over the call on the line.

TAPI.DLL can use the returned device ID with the corresponding media API to query the device's capabilities and subsequently open the media device. For example, if TAPI.DLL's client application needs to use the line as a waveform device, it first needs to call **waveInGetDevCaps** and/or **waveOutGetDevCaps** to determine the waveform capabilities of the device. The typical wave form

data format supported by telephony in North America is 8-bit m-law at 8000 samples per second, although the wave device driver can convert this sample rate and companding to other more common multimedia audio formats.

To subsequently open a line or address for audio playback using the waveform API, an application calls the **waveOutOpen** function. The implementation of the **waveOutOpen** call is device-specific, and each implementation has a number of options for implementing this function. If the call has not been answered yet when invoking the function, for example, the **waveOutOpen** driver may return an error code message to the application allowing it to display a dialog box stating that “the phone is ringing.” This is similar to requesting a CD to play back audio when no compact disc is in the player, and the existing definition of the waveform APIs should handle the situation.

Digit Monitoring

Digit monitoring monitors the call for digits. The SPI provides for digits to be signaled according to two methods (digit modes):

- **Pulse** Digits are signaled as pulse/rotary sequences. Note that for detection, these pulses manifest themselves as nothing more than sequences of audible clicks. Valid pulse digits are ‘0’ through ‘9’.
- **DTMF** Digits are signaled as DTMF (Dual Tone Multiple Frequency) tones. Valid DTMF digits are ‘0’ through ‘9’, ‘A’. ‘B’, ‘C’, ‘D’, ‘*’, and ‘#’. Both the beginning and the down edge of DTMF digits can be monitored.

Digit monitoring can be enabled or disabled on a specified call with **TSPI_lineMonitorDigits**. If enabled, all detected digits cause TAPI.DLL to be notified in the **LINE_MONITORDIGITS** callback message. The callback provides the call handle on which the digit was detected as well as the digit value and the digit mode. The scope of digit monitoring is bound by the lifetime of the call. Digit monitoring on a call ends as soon the call disconnects or goes idle.

Tone Monitoring

Tone monitoring monitors the media stream of a call for specified tones. A tone is described by its component frequencies and cadence. An implementation of the SPI may allow several different tones to be monitored simultaneously. The TSPI uses a two-level identification scheme to identify the tones monitored. There is a “tone group” and an “application-defined tag field” identifying a specific tone within a group.

TAPI.DLL can request monitoring for multiple concurrent tone groups, which the service provider should support if it has sufficient resources. Groups are monitored or canceled as a group—there is no finer granularity. Tone monitoring is enabled/disabled on a per-call basis using **TSPI_lineMonitorTones**.

When tones are detected, the **LINE_MONITORTONE** callback message notifies TAPI.DLL. This message includes the tone-group identifier and the application-specific identifier within that tone group to identify the tone that was detected. An application can “tag” each tone in order to be able to distinguish the different tones for which it requests detection. The scope of tone monitoring is bound by the lifetime of the call. Tone monitoring on a call ends as soon the call disconnects or goes idle.

Note that the monitoring of tones, digits, or media modes often requires the use of resources of which the service provider may only have a finite amount. A request for monitoring may be rejected if resources are not available. For the same reason, client applications should disable any monitoring that is not needed.

Media Control

TAPI.DLL can request the execution of a limited set of media control operations on the call's media stream triggered by telephony events. Although TAPI.DLL's clients normally use the media API specifically defined for the media mode, media control can yield a significant performance improvement for client/server implementations since simple "detect/control" sequences can be offloaded to the server. The operation **TSPI_lineSetMediaControl** allows TAPI.DLL to specify a list of tuples specifying a telephony event and the associated media-control action, and thus sets up a call's media stream for media control. The telephony events that can trigger media control activities are:

- **Detection of a digit** TAPI.DLL provides a list of specific digits and the media control action each of them triggers.
- **Detection of a media mode** TAPI.DLL provides a list of media modes and the media control actions a transition into the media mode triggers.
- **Detection of a specified tone** TAPI.DLL specifies a list of tones and the media control action each tone detection triggers.
- **Detection of a call state** TAPI.DLL specifies a list of call states and the media control action each transition to the call state triggers.

The media control actions listed below are defined generically for the different media modes. Not all media streams may provide meaningful interpretations of the media control actions. The operations should map well to audio streams.

- **Start** Starts the media stream.
- **Reset** Resets the media stream.
- **Pause** Stops or pauses the media stream.
- **Resume** Starts or resumes the media stream.
- **Rate up** Increases the rate (speed) of the media stream by a implementation-defined amount.
- **Rate down** Decreases the rate (speed) of the media stream by a implementation-defined amount.
- **Rate normal** Returns the rate (speed) to normal.
- **Volume up** Increases the volume (amplitude) of the media stream.
- **Volume down** Decreases the volume (amplitude) of the media stream.
- **Volume normal** Returns the volume (amplitude) to normal.

The scope of media control is bounded by the lifetime of the call. Media control on a call ends as soon the call disconnects or goes idle. Only a single media control request can be outstanding on a call across all applications.

Digit Gathering

Besides enabling digit monitoring and being notified of digits one at a time, TAPI.DLL can also request that multiple digits be collected in a buffer. Only when the buffer is full or when some other termination condition is met is TAPI.DLL notified. Digit gathering is useful for functions such as credit card number collection. TAPI.DLL invokes **TSPI_lineGatherDigits**, specifying a buffer to fill with digits. Digit gathering terminates when one of a number of conditions is true.

- The requested number of digits has been collected.
- One of multiple termination digits is detected. The termination digits are specified to **TSPI_lineGatherDigits**, and the termination digit is placed in the buffer as well.
- One of two timeouts expires. The timeouts are a first digit timeout, specifying the maximum duration before the first digit must be collected, and an inter-digit timeout, specifying the maximum duration between successive digits.
- Digit gathering is canceled explicitly by invoking **TSPI_lineGatherDigits** again with either another set of parameters (as with a new buffer) to start a new gathering request or by using a NULL digit buffer parameter to simply cancel.

When terminated for whatever reason, a **LINE_GATHERDIGITS** message is sent to the application that requested the digit gathering. Note that only a single digit gathering request can be outstanding on a call at any given time.

Note that digit gathering and digit monitoring may be enabled on the same call at the same time. In that case, TAPI.DLL receives a **LINE_MONITORDIGITS** message for each detected digit and a separate **LINE_GATHERDIGITS** message when the buffer is sent back.

Generating Inband Digits and Tones

Once in the *connected* state, information can be transmitted over a call. Two functions are provided that allow end-to-end inband signaling between TAPI.DLL and remote station equipment such as an answering machine. One function is **TSPI_lineGenerateDigits**, which signals digits over the voice channel on a call. Digits can be signaled as either rotary/pulse sequences or as DTMF tones. The other function is **TSPI_lineGenerateTone**, which enables TAPI.DLL to generate one of a variety of multi-frequency tones over the media stream on a call. This generates telephony tones, such as ringback, beep, busy, as well as arbitrary multi-frequency multi-cadenced tones.

Only one digit or tone generation can be in progress on a call at any one time. When digit or tone generation completes, a **LINE_GENERATE** message is sent to TAPI.DLL. In the case of generation of multiple digits, only a single message is sent back after all digits have been generated. Calling **TSPI_lineGenerateDigits** or **TSPI_lineGenerateTone** while digit or tone generation is in progress aborts the generation currently in progress and sends the **LINE_GENERATE** message to TAPI.DLL with a *cancel* indication.

Call Accept, Reject, and Redirect

In environments like ISDN, call offering is separate from alerting. In fact, after a call has been offered to TAPI.DLL, a time window exists during which TAPI.DLL has a number of options:

- TAPI.DLL can simply answer the call using **TSPI_lineAnswer**.
- TAPI.DLL can accept the call using **TSPI_lineAccept**, which also initiates alerting to the caller (as *ringback*), and to the called party (as *ring*).
- TAPI.DLL can reject the offering call using **TSPI_lineDrop**. The call remains valid but transitions to the *idle* state.
- TAPI.DLL can redirect the call using **TSPI_lineRedirect**, which deflects the offering call to another address. The call remains valid but transitions to the *idle* state.

Call Hold

Most PBXs can associate multiple calls with a single line. A call can be placed on *hard hold*. This frees up the user's line/address to make other calls. TAPI.DLL can place a call on hard hold by invoking **TSPI_lineHold**. TAPI.DLL may retrieve a call on hold by invoking **TSPI_lineUnhold**.

Hard hold is different from a *consultation hold*. A call is automatically placed on consultation hold, for example, when a call is prepared for transfer or conference.

Call Transfer

The Telephony SPI provides two mechanisms for call transfer: blind transfer and consultation transfer.

- In blind transfer (or single-step transfer), an existing call is transferred to a specified destination address in one phase using **TSPI_lineBlindTransfer**. This function transfers a call that was set up for transfer to another call.
- In a consultation transfer, the existing call is first prepared for transfer using **TSPI_lineSetupTransfer**. This places the existing call on consultation hold, and identifies the call as the target for the next transfer completion request. **TSPI_lineSetupTransfer** also allocates a consultation call that can be used to establish the consultation call with the party to be transferred to. TAPI.DLL can dial the extension of the destination party on the consultation call (using **TSPI_lineDial**), or it can drop and deallocate the consultation call and instead activate an existing held call (using **TSPI_lineUnhold**), if supported by the switch.

While the initial call is on consultation hold and the consultation call is active, TAPI.DLL can toggle between these calls with **TSPI_lineSwapHold**. It swaps the active call with the call currently on consultation hold.

Finally, TAPI.DLL completes the transfer in one of two ways using **TSPI_lineCompleteTransfer**:

- Transfer the call on transfer hold to the destination party. Both calls are cleared from the line—they transition to the *idle* state, but they remain valid.
- Enter a three-way conference. A new call handle is created to represent the conference and this handle is returned to TAPI.DLL.

Call Conference

Conference calls are calls that include more than two parties simultaneously. They can be set up using either a switch-based conference bridge or an external server-based bridge. Typically, only switch-based conferencing allows the level of conference control provided by the SPI. In server-based conference calls, all participating parties dial into the server, which mixes the media streams together and sends each participant the mix. There may be no notion of individual parties in the conference call, only that of a single call between the application and the bridge server.

A conference call can be established in a number of ways, depending on device capabilities:

- A conference call may be able to start out as a regular two-party call, that is, a call established with **TSPI_lineMakeCall**. Once the two-party call exists, additional parties can be added, one at a time. Preparing to add the third party establishes the conference call using **TSPI_lineSetupConference**. This operation takes the original two-party call as input, allocates a conference call, connects the original call to the conference, and allocates a consultation call whose handle is returned to TAPI.DLL.

  TAPI.DLL can then use **TSPI_lineDial** on the consultation call to establish a connection to the next party to be added. **TSPI_lineDrop** can be used to abandon this call attempt. The third party is added with **TSPI_lineAddToConference**, which specifies both the conference call and the consultation call.
- A three-way conference call can be established by resolving a transfer request for a three-way conference. In this scenario, a two-party call is established as either an inbound or outbound call. Next the call is placed on transfer hold with **TSPI_lineSetupTransfer** which returns a consultation call handle. After a period of consultation, TAPI.DLL may have the option to resolve the transfer setup by selecting the three-way conference option which conferences all three parties together in a conference call using **TSPI_lineCompleteTransfer** with the *conference* option (instead of the *transfer* option). Under this option, a conference call handle representing the conference call is allocated and returned to the application.
- A conference call may need to be established with **TSPI_lineSetupConference** without an existing two-party call. This returns a handle for the conference call, and allocates a consultation call. After a period of consultation, the consultation call can be added using **TSPI_lineAddToConference**. Additional parties are added with **TSPI_linePrepareAddToConference** followed by **TSPI_lineAddToConference**.

To add additional parties once a conference call exists, TAPI.DLL uses **TSPI_linePrepareAddToConference**, which specifies the conference call handle and returns a consultation call handle. Once the consultation call exists, it can be added using **TSPI_lineAddToConference** (as described earlier).

Once a call becomes a member of a conference call, the member's call state reverts to *conferenced*. The state of the conference call typically becomes *connected*. The call handle to the conference call and all the added parties remain valid as individual calls. **LINE_CALLSTATE** events can be received about all calls, for example, if one of the members disconnects by hanging up, an appropriate call state message may inform the application of this fact; such a call is no longer a member of the conference.

As is the case with call transfer, the application may toggle between the consultation call and the conference call using **TSPI_lineSwapHold**.

TAPI.DLL may use the call handle for the member calls to later remove the call from the conference, and it does this by invoking **TSPI_lineRemoveFromConference** on the call handle. Note that this operation is not commonly available in its fully general form. Some switches may not allow it at all, or only allow the most recently added party to be removed. The line's device capabilities describe which type of **TSPI_lineRemoveFromConference** is possible.

Removing a Party

When canceling the consultation call to the third party on a conference call or when removing the third party in a previously established conference call, the service provider (and switch) may release the conference bridge and revert the call back to a normal two-party call. If this is the case, the call whose handle is passed (in an invocation of the function **TSPI_linePrepareAddToConference**) with *hConfCall* transitions to the *idle* state, and the only remaining participating call transitions to the *connected* state.

Call Park

Two forms of call parking are provided: *directed* call park and *non-directed* call park. In directed call park, the application specifies the destination address where the call is to be parked. This roughly behaves like a call transfer to the destination address, but it doesn't alert or time out as a transfer would.

In non-directed call park, the switch returns to TAPI.DLL the address where it parked the call. In either case, the function **TSPI_linePark** is used to park a given call at another address.A parked call can later be retrieved. TAPI.DLL specifies the park address to **TSPI_lineUnpark**, which returns a call handle to the unparked call (it retrieves a parked call). Appropriate **LINE_CALLSTATE** messages are sent to TAPI.DLL as the call is reconnected.

Call Forwarding

Forwarding affects the treatment by the switch or network of incoming calls destined for a given address. TAPI.DLL can specify call forwarding conditions based on origin of call (internal, external, selective based on caller ID), status of the address (busy, no answer, unconditionally), and destination address where calls are to be forwarded. When the specified conditions are met for an incoming call, the switch deflects the incoming call to the specified destination number. Note that because the switch performs the forwarding action, TAPI.DLL typically does not know when a call has been forwarded.

The **TSPI_lineForward** function provides a combination of call forwarding and do-not-disturb features. **TSPI_lineForward** also cancels any or all of the forwarding requests currently in effect. Some switches require that a call be established to the forwarding address in order for call forwarding to be initiated. On such systems, **TSPI_lineForward** allocates a consultation call and returns the handle for it to TAPI.DLL. The consultation call can be used as any other call. After the connection is established, forwarding confirmation is received from the switch, and the call is dropped (using **TSPI_lineDrop**), forwarding is in effect. A **LINE_ADDRESSSTATE** message with a *forwarding* indication informs TAPI.DLL about changes in an address' forwarding status.

Note that it may be impossible for a service provider to know at all times what forwarding is in effect for an address. Forwarding may be canceled or changed in ways that make it impossible for a service provider to be informed of this fact.

Call Pickup

Call pickup allows TAPI.DLL to answer a call that is alerting at another address. TAPI.DLL invokes **TSPI_linePickup** by identifying the target of the pickup and is returned a call handle for the picked up call. (Its function is to pick up a call that is alerting at another number.) There are several ways to specify the target of the pickup request. First, specify the address (extension) of the alerting party. Second, if no extension is specified and the switch allows it, TAPI.DLL can pick up any ringing phone in its pickup group. Third, some switches require a group ID to identify the group to which the ringing extensions belongs.

After the call has been picked up, the call is diverted to TAPI.DLL, and TAPI.DLL is also sent appropriate **LINE_CALLSTATE** messages for the call. TAPI.DLL can invoke **TSPI_lineGetCallInfo** to find out information about the picked up call, if provided by the switch.

Call Completion

When making an outbound call, the unavailability of certain resources may prevent the call from reaching the *connected* state, as when the destination party is busy or doesn't answer. Unavailable resources include trunk circuits as well as the destination party's station. **TSPI_lineCompleteCall** places a call-completion request in that it allows TAPI.DLL to specify how its wants to complete a call that cannot be completed normally. TAPI.DLL has the following options:

- **Camp on** to queue the call until the call can be completed.
- **Call back** requests the called station to return the call when it returns to idle. Answering the call back may automatically re-initiate (redial) the connection request.
- **Intrude** allows TAPI.DLL to barge in on the existing call.
- **Message** (also known as "leave word calling") allows TAPI.DLL to send one of a small number of predefined messages to the destination. These messages can be text shown on the phone's display, a voice message left for the user, and so on.

A call completion request can be canceled with **TSPI_lineUncompleteCall**. Multiple call completion request can potentially be outstanding for a given address at any one time. To identify individual requests, the implementation returns a completion ID. When a call completion request completes and results in a new call, the call completion ID is available in the **LINECALLINFO** data structure returned by **TSPI_lineGetCallInfo**. Canceling a call completion request in progress also uses this call completion ID.

Extended Line Services

Extended Line Services (or device-specific line services) include all service-provider defined extensions to the SPI. The SPI defines a mechanism that enables service-provider vendors to extend the Telephony SPI using device-specific extensions. The SPI only defines the extension mechanism, and by doing so provides access to device-specific extensions, but the SPI does not define their behavior. Behavior is completely defined by the service provider.

The Telephony SPI consists of scalar and bit flag constant definitions, data structures, functions, and callback messages. Procedures are defined that enable a vendor to extend most of these as follows:

For extensible scalar data constants, a service-provider vendor may define new values in a specified range. As most data constants are DWORDs, typically the range 0x00000000 through 0x7FFFFFFF is reserved for common future extensions, while 0x80000000 through 0xFFFFFFFF are available for vendor-specific extensions. The assumption is that a vendor would define values that are natural extensions of the data types defined by the SPI.

For extensible bit flag data constants, a service-provider vendor may define new values for specified bits. As most bit flag constants are DWORDs, typically a specific number of the lower bits are reserved for common extensions while the remaining upper bits are available for vendor-specific extensions. Common bit flags are assigned from bit zero up; vendor-specific extensions should be assigned from bit 31 down. This provides maximum flexibility in assigning bit positions to common extensions versus vendor-specific extensions. A vendor is expected to define new values that are natural extensions of the bit flags defined by the SPI.

Extensible data structures have a variable sized field that is reserved for device-specific use. Being variable sized, the service provider decides the amount of information and the interpretation. A vendor that defines a device-specific field is expected to make these natural extensions of the original data structure defined by the SPI.

Two functions, **TSPI_lineDevSpecific** and **TSPI_lineDevSpecificFeature**, and four related messages, **LINE_DEVSPECIFIC, LINE_CALLDEVSPECIFIC, LINE_DEVSPECIFICFEATURE,** and **LINE_CALLDEVSPECIFICFEATURE**, offer a vendor-specific extension mechanism. The **TSPI_lineDevSpecific** operation and associated **LINE_DEVSPECIFIC** and **LINE_CALLDEVSPECIFIC** messages allow TAPI.DLL's client application to access device-specific line, address, or call features that are unavailable in the Basic or Supplementary Telephony Services. The parameter profile of the **TSPI_lineDevSpecific** function is generic in that little interpretation of the parameters is made by the SPI. Device-handle parameters have TSPI-defined meanings and are translated appropriately between the application and the service provider. Generic parameters are simply passed through unmodified. The interpretation of the generic parameters is defined by the service provider and must be understood by any applications that use them. An application that relies on device-specific extensions generally does not work with other service providers. However, applications written entirely to the Basic and Supplementary Telephony services should work with the extended service provider.

TAPI.DLL's implementation of the device-specific functions and messages is "pass-through." TAPI.DLL does not examine or modify the "generic" device-specific parameters and buffers, but it does map application-level opaque handle values (used at the TAPI level) to service-provider-level opaque handle values (used at the TSPI level).

Regarding handle translation, the pass-through nature of the generic parts of device-specific extensions has an imporant consequence. A service provider has no way to relate the handles used at the TSPI level to those at the TAPI level except by passing them through the predefined handle parameters and fields. Any handle put into the generic extension area is untranslated by TAPI.DLL as it is passed between application and service provider. The designer of a service-provider extension should generally not define extensions that pass handles in this way.

The appropriate approach when defining a device-specific extension that needs to refer to specific devices without using handles is to refer to them using their absolute device identification. The device ID used in opening a line at the TAPI level, for example, is strictly the same value that is used at the TSPI level to open the line. Similarly, a (line device ID, address ID) tuple that uniquely identifies an address at the TAPI level uses the same values to identify the same thing at the TSPI level. This is the motivation for the **TSPI_lineSetDeviceIDBase** procedure. It adjusts the service provider's device ID numbering to be the same as the numbering used at the TAPI level. This gives the service-provider

designer the flexibility to use device IDs to identify devices in device-specific extensions.

For convenience, a more specialized escape function is also provided. It is similar to **TSPI_lineDevSpecific**, but places interpretation on some of the parameters. The function **TSPI_lineDevSpecificFeature** and associated **LINE_DEVSPECIFICFEATURE** and **LINE_CALLDEVSPECIFICFEATURE** messages allow TAPI.DLL to emulate button presses at the line's feature phone. As feature phones and the meanings of their buttons are vendor-specific, feature invocation using **TSPI_lineDevSpecificFeature** is also vendor-specific.

To summarize, the function **TSPI_lineDevSpecificFeature** is a device-specific escape function to allow sending switch features to the switch. The message **LINE_CALLDEVSPECIFICFEATURE** is a device-specific message sent to the application's callback as an indication of call-related features sent to the switch. **LINE_DEVSPECIFICFEATURE** is a device-specific message sent to the application's callback as an indication of line-related features sent to the switch.

As mentioned earlier, there is no central registry for manufacturer IDs. Instead a unique ID generator is made available as part the Telephony SDK. The vendor that designs a set of device-specific extensions uses this utility to obtain a unique identifier for those extensions. This unique identifier must be published as part of the specification of the extensions to allow application writers to access the extensions.

Phone Devices

 About Phone Devices

Phone Devices

About Phone Devices

Introduction

Telephony defines a device that supports the phone device class as a device that includes some or all of the following elements:

- **Hookswitch/Transducer** This is a means for audio input and output. A phone device may have several transducers, which can be activated and deactivated (taken offhook or placed onhook) under application or manual user control. Telephony identifies three types of hookswitch devices common to many phone sets:

  Handset The traditional mouth-and-ear piece combination that must be manually lifted from a cradle and held against the user's ear.

  Speakerphone Enables the user to conduct calls hands-free. The speakerphone may be internal or external to the phone device. The speaker part of a speakerphone allows multiple listeners.

  Headset Enables the user to conduct calls hands-free.

  A hookswitch must be offhook to allow audio data to be sent to and/or received by the corresponding transducer.
- **Volume Control/Gain Control/Mute** Each hookswitch device is the pairing of a speaker and a microphone component. Telephony provides for volume control and muting of speaker components and for gain control or muting of microphone components.
- **Ringer** A means for alerting users, usually through a bell. A phone device may be able to ring in a variety of modes or patterns.
- **Display** A mechanism for visually presenting messages to the user. A phone display is characterized by its number of rows and columns.
- **Buttons** An array of buttons. Whenever the user presses a button on the phone set, a report is generated indicating that button press. Button-Lamp IDs identify a button and lamp pair. Of course, it is possible to have button-lamp pairs with either no button or no lamp. Button-lamp IDs are integer values that range from 0 to the maximum number of button-lamps available on the phone device, minus one. Each button belongs to a button class. Classes include call appearance buttons, feature buttons, keypad buttons, and local buttons.
- **Lamps** An array of lamps (such as LEDs) that are individually controllable. Lamps can be lit in different modes by varying the on and off frequency. The button-lamp ID identifies the lamp.
- **Data Areas** Memory areas in the phone device where instruction code or data can be downloaded to and/or uploaded from. The downloaded information would affect the behavior (or in other words, program) the phone device.

Telephony allows an application to monitor and control elements of the phone device. The most useful elements for an application are the hookswitch devices. The phone set can act as an audio I/O device (to the computer) with volume control, gain control and mute, a ringer (for alerting the user), data areas (for programming the phone), and perhaps a display, though the computer's display is more capable. The application writer is discouraged from directly controlling or using phone lamps or phone buttons, since lamp and button capabilities can vary widely among phone sets, and applications can quickly become tailored to specific phone sets.

There is no guaranteed core set of services supported by all phone devices as there is for line devices (the Basic Telephony services). Therefore, before a phone device can be used, its exact capabilities must first be determined. Telephony capability varies with the configuration (client versus client-server), the telephone hardware, and the service-provider software. Applications should make no assumptions as to what telephony capabilities are available.

The capabilities of a phone device are determined by a call to the **TSPI_phoneGetDevCaps** function. This function queries a specified phone device to determine its telephony capabilities. With a call to this function, the service provider should fill in all the fields of the PHONEBUTTONINFO data structure except **dwTotalSize**, which is filled in by TAPI.DLL. The service provider must not overwrite the **dwTotalSize** field.

A phone's device capabilities indicate which of these elements exist for each phone device present in the system and what their capabilities are. Although strongly oriented towards real-life telephone sets, this abstraction can provide a meaningful implementation (or subset thereof) for other devices as well. Take as an example a separate headset directly connected and controllable from the computer and operated as a phone device. Hookswitch changes can be triggered by detection of voice energy

(offhook) or a period of silence (onhook); ringing can be emulated by the generation of an audible signal into the headset; a display can be emulated through text-to-speech conversion.

Note that a phone device need not be realized in hardware, but can instead be emulated in software using a mouse- or keyboard-driven graphical command interface and the computer's speaker or sound system. Such a "soft phone" can be an application that uses TAPI. It can also be a service provider, which can be listed as a phone device available to other applications through the API, and as such is assigned a phone device ID.

Basic Phone Services

Introduction: Basic Phone Services

Phone SPI Initialization

As part of the phone device abstraction defined by TSPI, TAPI.DLL and the service provider must first undergo basic initialization. This basic initialization is accomplished both for the line and phone halves of the interface by the same set of steps. The first of these steps is interface version negotiation. TAPI.DLL performs this by calling the **TSPI_lineNegotiateTSPIVersion** function. This function is usually used to negotiate on behalf of an individual line device; different line devices within the same service provider may operate according to different interface versions. TAPI.DLL passes a special reserved device ID value, **INITIALIZE_NEGOTIATION,** to indicate that it is negotiating an overall interface version for initialization functions that affect the entire service provider, both for lines and phones.

The result of this negotiation is passed to subsequent procedures until a phone device is opened. At that time, the phone device becomes committed to a particular interface version. This interface version is implicit until the phone is closed, and does not need to be passed to subsequent functions that operate on an opened phone.

Following overall interface version negotiation, TAPI.DLL calls the function **TSPI_providerInit**. This function initializes the service provider, also giving it parameters required for subsequent operation. These parameters include the following:

dwPermanentProviderID specifies the permanent ID, unique within the service providers on this system, of the service provider being initialized.

dwLineDeviceIDBase specifies the lowest device ID for the line devices supported by this service provider.

dwPhoneDeviceIDBase specifies the lowest device ID for the phone devices supported by this service provider. Devices of the Telephony “phone” device class are identified by integers starting from zero. This range of identifiers is contiguous across the full range of phone devices. Since there may be multiple service providers managing phone devices in a single system, each service provider gets a contiguous portion of the total range. This parameter tells the service provider the lowest value in its portion of the range. The service provider, rather than TAPI.DLL, has the responsibility for mapping this variable range to its own internal device identifiers. This gives the service-provider vendor sufficient flexibility to use device IDs in device-specific extensions if it so desires. Since the service provider “knows” what device IDs appear in the TAPI-defined parameters and data structures, it can use consistent device IDs in device-specific extension parameters and data structures.

dwNumLines specifies how many line devices this service provider supports.

dwNumPhones specifies how many phone devices this service provider supports.

lpfnCompletionProc specifies the procedure the service provider calls to report completion of all asynchronously operating procedures on line and phone devices.

Following **TSPI_providerInit**, normal operations such as opening phones can be carried out.

Phones

The Telephony SPI allows TAPI.DLL to monitor and/or control elements of the phone device on behalf of its client applications. Probably the most useful elements for an application to use are the hookswitch devices (as when the phone set is used as an audio I/O device to the PC) with volume control, gain control and mute, the ringer (for alerting the user), the data areas (for programming the phone), and perhaps the display (the PC's display is certainly a lot more capable). The application writer is discouraged from directly controlling or using phone lamps or phone buttons, since lamp and button capabilities can vary widely among phone sets, and applications can quickly become tailored to specific phone sets.

There is no guaranteed core set of services supported by all phone devices as there is for line devices with the Basic Telephony Services. Therefore, before an application can use a phone device, the application must first determine the exact capabilities of the phone device. Telephony capability varies with the configuration (client versus client/server, for example), the telephone hardware, and the service-provider software. Applications should make no assumptions as to what telephony capabilities are available. TAPI.DLL determines a phone's device capabilities on behalf of an application using **TSPI_phoneGetDevCaps**.

A phone's device capabilities indicate which of these elements exist for each phone device present in the system and what their capabilities are. Although strongly oriented towards real-life telephone sets, this abstraction can provide a meaningful implementation (or subset thereof) for other devices as well. Take as an example a separate headset directly connected and controllable from the PC and operated as a phone device. Hookswitch changes can be triggered by detection of voice energy (offhook) or a period of silence (onhook). Ringing can be emulated by the generation of an audible signal into the headset. A display can be emulated by means of text-to-speech conversion.

Note that a phone device need not be realized in hardware, but can instead be emulated in software, using a mouse- or keyboard-driven graphical command interface and the PC's speaker or sound system. Such a "soft phone" can be an application of the Telephony API. It can also be a service provider, which can be listed as a phone device available to applications through the SPI (that is, it is assigned a phone device ID).

Depending on the environment and configuration, phone sets can be shared devices between the application and the switch. Some minor provision is made in the SPI where the switch may temporarily suspend the SPI's control of a phone device (see further).

Opening and Closing Phone Devices

After having performed basic initialization and possibly having retrieved device capabilities, TAPI.DLL must open the phone device before it can access functions on that phone. After a phone device has been successfully opened, TAPI.DLL is returned a handle representing the open phone. The function **TSPI_phoneOpen** opens the specified phone device prior to providing access to functions on the phone. **TSPI_phoneClose** closes a specified phone device.

A phone device is identified to **TSPI_lineOpen** by means of its device ID. This function returns a handle to an open phone. A handle to an open phone device is used in other operations to identify the open phone device. Note that the only functions on phone devices that take a phone device ID parameter are the early initialization functions such as **TSPI_phoneGetDevCaps**, **TSPI_phoneNegotiateTSPIVersion**, **TSPI_phoneNegotiateExtVersion**, and **TSPI_phoneOpen** functions. All other functions take phone handles.

Given a phone handle, the phone's device ID can be retrieved with the function **TSPI_phoneGetID**.

TAPI.DLL's client application can obtain device IDs for various Windows device classes associated with an opened phone by invoking **TSPI_phoneGetID** by using the TAPI function **phoneGetID**. This function takes a phone handle and a device class description. It returns the device ID for the device of the given device class that is associated with the open phone device. If the device class is "tapi/phone," the device ID of the phone device is returned. If the device class is "mci/wave," the device ID of an mci waveaudio device (if supported) is returned that allows the manipulation (such as recording or playback) of audio over the phone.

To subsequently open a phone device for audio playback using the waveform API, an application calls the **waveOutOpen** function. The implementation of the **waveOutOpen** call is device-specific, and each implementation has a number of options for implementing this function. If the phone's handset is on hook when invoking the function, for example, the **waveOutOpen** driver may return an error code message to the application that is further mapped in a request (dialog box) to the user to "pick up the phone." This is similar to requesting a CD to play back audio when no compact disc is in the player, and the existing definition of the waveform APIs can deal with it.

In contrast with line devices, for which the basic line services provide the equivalent of POTS, no minimum guaranteed set of functions is defined for phone devices. While each phone device provides at least the functions and messages listed in this section, these operations only provide a way for the software to refer to the phone; they do not offer any true operations on the physical phone device.

Supplementary Phone Services

Introduction: Supplementary Phone Services

Hookswitch Devices

A phone device may have multiple hookswitch devices. Each hookswitch device has a speaker and a microphone component, and operates in one of four hookswitch modes:

- **Onhook** The hookswitch device is onhook, and both its microphone and speaker are disabled.
- **Microphone only** The hookswitch device is offhook, its microphone is enabled, and its speaker is mute.
- **Speaker only** The hookswitch device is offhook, its microphone is disabled and its speaker is enabled.
- **Microphone and speaker** The hookswitch device is offhook, both the microphone and the speaker are enabled.

TAPI.DLL can set and query hookswitch mode of an opened phone's hookswitch devices. To mute/unmute the microphone or speaker component of a hookswitch device, the operation **TSPI_phoneSetHookSwitch** is used with the appropriate hookswitch mode. This function sets the hookswitch mode of one or more of the hookswitch devices of an open phone device. The function **TSPI_phoneGetHookSwitch** queries the hookswitch mode of a hookswitch device of an open phone device.

When the mode of a phone's hookswitch device is changed manually, for example by lifting the handset from its cradle, the service provider must send a **PHONE_STATUS** message to TAPI.DLL's callback function to notify it about the state change. Parameters to this message provide an indication of the change.

The volume of the speaker component of a hookswitch device can be set with **TSPI_phoneSetVolume**. Note that volume setting is different from mute in that muting a speaker and later unmuting it preserves the volume setting of the speaker. The **TSPI_phoneGetVolume** function returns the volume setting of a hookswitch device's speaker of an open phone device.

The microphone component of a hookswitch device can be gain controlled. Note that gain setting is different from mute in that muting a microphone and later unmuting it preserves the gain setting of the microphone. **TSPI_phoneSetGain** sets the gain of a hookswitch device's microphone of an open phone device, and **TSPI_phoneGetGain** returns the gain setting of a hookswitch device's microphone of an opened phone.

When the volume or gain of a phone's hookswitch device is changed, the service provider sends a **PHONE_STATUS** message to TAPI.DLL's callback function to notify TAPI.DLL about the state change. Parameters to this message provide an indication of the change.

Display

The SPI provides access to a phone's display. The display is modeled as an alphanumeric area with rows and columns. A phone's device capabilities indicate the size of a phone's display as number of rows and number of columns. Both these number are zero if the phone device does not have a display. The function **TSPI_phoneSetDisplay** writes information to the display of an open phone device. **TSPI_phoneGetDisplay** returns the current contents of a phone's display.

When the display of a phone device is changed, the service provider sends a **PHONE_STATUS** message to TAPI.DLL's callback function to notify it about the state change. Parameters to this message provide an indication of the change.

Ring

A single phone may be able to ring with different ring modes. Given the wide variety of ring modes available, ring modes are identified by their ring mode number. A ring mode number ranges from zero to the number of available ring modes, with ring mode zero indicating silence.

TSPI_phoneSetRing rings an open phone device according to a given ring mode.**TSPI_phoneGetRing** returns the current ring mode of an opened phone device.

When the ring mode of a phone device is changed, the service provider sends a **PHONE_STATUS** message to TAPI.DLL's callback function to notify it about the state change. Parameters to this message provide an indication of the change.

Buttons

Windows Telephony models a phone's buttons and lamps as button-lamp pairs. A button with no lamp next to it, or a lamp with no button is specified using a "dummy" indicator for the missing lamp or button. A button with multiple lamps is modeled by using multiple button-lamp pairs.

Information associated with a phone button can be set and retrieved. When a button is pressed, the service provider sends a **PHONE_BUTTON** message to TAPI.DLL's callback function. Parameters of this message are a handle to the phone device and the button/lamp ID of the button that was pressed. The keypad button '0' through '9', '*', and '#' are assigned fixed button/lamp IDs 0 through 11.

The function **TSPI_phoneSetButtonInfo** sets the information associated with a button on a phone device. **TSPI_phoneGetButtonInfo** returns information associated with a button on a phone device. The service provider sends a **PHONE_BUTTON** message to TAPI.DLL's callback function when a button on the phone is pressed.

The information associated with a button does not carry any semantic meaning as far as the SPI is concerned. It is useful for device-specific applications that understand the meaning of this information for a given phone device, or for display to the user, such as in online help.

Lamps

The lamps on a phone device can be lit in a variety of different lighting modes. Unlike ringing patterns, lamp modes are more uniform across phone sets of different vendors. A common set of lamp modes is defined by Windows Telephony. A lamp identified by its lamp/button ID can be lit in a given lamp mode. A lamp can also be queried for its current lamp mode.

The lamp functions are **TSPI_phoneSetLamp**, which lights a lamp on a specified open phone device in a given lamp lighting mode, and **TSPI_phoneGetLamp**, which returns the current lamp mode of the specified lamp.

When a lamp of a phone device is changed, the service provider sends a **PHONE_STATUS** message to TAPI.DLL's callback function to notify it about the state change. Parameters to this message provide an indication of the change.

Data Areas

Some phone sets support the notion of downloading data from or uploading data to the phone device, which allows the phone set to be programmed in a variety of ways. Windows Telephony models these phone sets as having one or more download or upload areas. Each area is identified by a number that ranges from zero to the number of data areas available on the phone minus one. Sizes of each area may vary. The format of the data itself is device specific.

The function **TSPI_phoneSetData** downloads a buffer of data to a given data area in the phone device. **TSPI_phoneGetData** uploads the contents of a given data area in the phone device to a buffer.

When a data area of a phone device is changed, the service provider sends a **PHONE_STATUS** message to TAPI.DLL's callback function to notify it about the state change. Parameters to this message provide an indication of the change.

Status

Most of the **Get** and **Set** operations described earlier deal with one component of information only. The operation **TSPI_phoneGetStatus** allows TAPI.DLL to obtain complete status information about a phone device.

As mentioned earlier, whenever a status item changes on the phone device, the service provider sends a **PHONE_STATUS** message to TAPI.DLL's callback function. The message parameters include a handle to the phone device and an indication of the status item that changed. TAPI.DLL can use **TSPI_phoneSetStatusMessages** to select status changes for which it wants to be notified in the callback message. In contrast to the TAPI interface, there is no corresponding **TSPI_phoneGetStatusMessages** function. TAPI.DLL keeps track of which applications have enabled which status messages by itself.

Extended Phone Services

The Extended Phone Services (or device-specific phone services) include all extensions to the SPI defined by the service-provider. The SPI defines a mechanism that enables service-provider vendors to extend the Telephony SPI using device-specific extensions. The SPI only defines the extension mechanism, and by doing so provides access to device-specific extensions, whose behavior is completely defined by the service provider.

The Telephony SPI consists of scalar and bit flag data constant definitions, data structures, functions, and callback messages. Procedures are defined that enable a vendor to extend most of these as follows:

Scalar Data Constants

For extensible scalar data constants, a service-provider vendor may define new values in a specified range. As most data constants are DWORDs, typically the range 0x00000000 through 0x7FFFFFFF is reserved for common future extensions, while 0x80000000 through 0xFFFFFFFF are available for vendor-specific extensions. The assumption is that a vendor would define values that are natural extensions of the data types defined by the SPI.

Bit-Flag Data Constants

For extensible bit flag data constants, a service-provider vendor may define new values for specified bits. As most bit flag constants are DWORDs, typically a specific number of the lower bits are reserved for common extensions while the remaining upper bits are available for vendor-specific extensions. Common bit flags are assigned from bit zero up, and vendor-specific extensions should be assigned from bit 31 down. This provides maximum flexibility in assigning bit positions to common extensions versus vendor-specific extensions. A vendor is expected to define new values that are natural extensions of the bit flags defined by the SPI.

Extensible data structures have a variable sized field that is reserved for device-specific use. Being variable sized, the service provider decides the amount of information and the interpretation. A vendor that defines a device-specific field is expected to make these natural extensions of the original data structure defined by the SPI.

Functions and Messages

The **TSPI_phoneDevSpecific** operation and associated **PHONE_DEVSPECIFIC** message enable TAPI.DLL's client application to access device-specific phone features that are unavailable through the common Telephony Services on phones. The parameter profile of the **TSPI_phoneDevSpecific** function is generic in that little interpretation of the parameters is made by the SPI. Device-handle parameters have TSPI-defined meanings and are translated appropriately between the application and the service provider. Generic parameters are simply passed through unmodified. The interpretation of the generic parameters is defined by the service provider and must be understood by any applications that use them. An application that relies on device-specific extensions will not generally work with other service providers. However, applications written entirely to the common telephony phone services will work with the extended service provider.

Regarding handle translation, the pass-through nature of the generic parts of device-specific extensions has an imporant consequence. A service provider has no way to relate the handles used at the TSPI level to those at the TAPI level except by passing them through the predefined handle parameters and fields. Any handle put into the generic extension area is untranslated by TAPI.DLL as it is passed between application and service provider. The designer of a service-provider extension should generally not define extensions that pass handles in this way.

The appropriate approach when defining a device-specific extension that needs to refer to specific devices without using handles is to refer to them using their absolute device identification. The device ID used in opening a phone at the TAPI level, for example, is strictly the same value that is used at the TSPI level to open the phone.

The appropriate approach when defining a device-specific extension that needs to refer to specific devices without using handles is to refer to them using their absolute device identification. The device ID used in opening a phone at the TAPI level, for example, is strictly the same value that is used at the TSPI level to open the phone. This is the motivation for the *dwPhoneDeviceIDBase* parameter in the **TSPI_providerInit** function. It adjusts the service provider's device ID numbering to be the same as

the numbering used at the TAPI level. This gives the service-provider designer the flexibility to use device IDs to identify devices in device-specific extensions.

As mentioned earlier, there is no central registry for manufacturer IDs. Instead a unique ID generator is made available as part the Telephony SDK. The vendor that designs a set of device-specific extensions uses this utility to obtain a unique identifier for those extensions. This unique identifier must be published as part of the specification of the extensions to allow application writers to access the extensions.

Service-Provider Functions

 Reference

Service-Provider Functions

 Reference

 Functions

 Constants

Reference

Functions

- TSPI_providerConfig
- TSPI_providerInit
- TSPI_providerInstall
- TSPI_providerRemove
- TSPI_providerShutdown

TSPI_providerConfig

[New - Windows 95]

```
LONG TSPI_providerConfig(HWND hwndOwner, DWORD dwPermanentProviderID)
```

Creates or modifies the [Provider<PPID>] section of the TELEPHON.INI file as needed for the service provider to operate. It may use a dialog box to gather configuration information from the user, and this dialog may include sub-dialogs associated with other APIs (such as Comm) for the setup of specific devices.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

  LINEERR_OPERATIONFAILED

  LINEERR_NOMEM

hwndOwner

Specifies the handle of the parent window in which the function may create any dialog windows required during the configuration.

dwPermanentProviderID

Specifies the service provider's permanent provider ID that must match the <PPID> of the [Provider<PPID>] section to be modified.

If this function returns success, it must guarantee that the NUMLINES and NUMPHONES settings in the [Provider x] section of TELEPHON.INI are valid. It should also write any service provider-specific information to the [Provider x] section.

This function may be called while the service provider is in use (that is, between calls of **TSPI_providerInit** and **TSPI_providerShutdown**).

Any changes that affect the behavior visible through the TSPI should take effect only when the service provider is restarted at the next **TSPI_providerInit**.

The Telephony Control Panel applet supplied with the Windows Telephony SDK calls this function when the "setup" command is invoked.

The Telephony Control Panel applet will also make the appropriate updates to the [Windows Telephony] section of the WIN.INI file and issue a WM_WININICHANGE message to notify TAPI.DLL of any changes made to this file. TAPI.DLL invokes **TSPI_providerShutdown** and waits until all Telephony operations have shutdown to guarantee that each service provider will read the new configuration the next time it is invoked.

There is no directly corresponding function at the TAPI level. At that level, applications have access to the functions **lineConfigDialog** and **phoneConfigDialog**, which allow configuration of parameters of a particular line or phone once it has been installed.

TSPI_providerInit

[New - Windows 95]

```
LONG TSPI_providerInit(DWORD dwTSPIVersion, DWORD dwPermanentProviderID,
DWORD dwLineDeviceIDBase, DWORD dwPhoneDeviceIDBase, DWORD dwNumLines, DWORD
dwNumPhones, ASYNC_COMPLETION lpfnCompletionProc)
```

Initializes the service provider, also giving it parameters required for subsequent operation. The service provider must discard any information it has acquired from reading TELEPHON.INI earlier, since this may have changed.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INCOMPATIBLEAPIVERSION	LINEERR_OPERATIONFAILED
LINEERR_NOMEM	LINEERR_RESOURCEUNAVAIL
LINEERR_INIFILECORRUPT	LINEERR_NOMULTIPLEINSTANCE

dwTSPIVersion

Specifies the version of the TSPI definition under which this function must operate. The caller may use **TSPI_lineNegotiateTSPIVersion** with the special *dwDeviceID* INITIALIZE_NEGOTIATION to negotiate a version that is guaranteed to be acceptible to the service provider.

dwPermanentProviderID

Specifies the permanent ID, unique within the service providers on this system, of the service provider being initialized. The service provider may search TELEPHON.INI for a [Provider<PPID>] section whose <PPID> matches this value.

dwLineDeviceIDBase

Specifies the lowest device ID for the line devices supported by this service provider.

dwPhoneDeviceIDBase

Specifies the lowest device ID for the phone devices supported by this service provider.

dwNumLines

Specifies how many line devices this service provider supports. This number is guaranteed to match the corresponding parameter in TELEPHON.INI

dwNumPhones

Specifies how many line devices this service provider supports. This number is guaranteed to match the corresponding parameter in TELEPHON.INI

lpfnCompletionProc

Specifies the procedure the service provider calls to report completion of all asynchronously operating procedures on line and phone devices.

It is the caller’s responsibility to ensure that service provider’s installation is complete before this function is called. The service provider's *ProviderIDx* entry with its <PPID> value matching the *dwPermanentProviderID* and the service provider's *ProviderFilenamex* entry have already been made in the [Providers] section. In addition, the [Provider<PPID>] section must have been created with the *NumLines* and *NumPhones* entries present.

This function is guaranteed to be called before any of the other functions prefixed with **TSPI_line** or **TSPI_phone** except **TSPI_lineNegotiateTSPIVersion**. It is strictly paired with a subsequent call to **TSPI_providerShutdown**. It is the caller's reponsibility to ensure proper pairing. These pairs may overlap, for example, when the Telephony Control Panel applet is used while Telephony operations are in progress. The call to this function must be ignored (returning success) if there is already an outstanding pair.

Note that a service provider should perform as many consistency checks as is practical at the time **TSPI_providerInit** is called to ensure that it is ready to run. Some consistency or installation errors, however, may not be detectable until the operation is attempted. The error LINEERR_NODRIVER can be used to report such non-specific errors at the time they are detected.

There is no directly corresponding function at the TAPI level. At that level, multiple different usage instances can be outstanding, with an “application handle” returned to identify the instance in subsequent operations. At the TSPI level, the interface architecture supports only a single usage instance for each distinct service provider.

TSPI_providerInstall

[New - Windows 95]

```
LONG TSPI_providerInstall(HWND hwndOwner, DWORD dwPermanentProviderID)
```

Installs any additional “pieces” of the provider into the right directories (or at least verifying that they’re there), sets up the provider’s TELEPHON.INI file entries for its lines and phones, and creates entries in other INI files (such as SYSTEM.INI, for drivers) as necessary. It is called from the Telephony Control Panel applet when the “Add” button is pressed.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

  LINEERR_OPERATIONFAILED LINEERR_INIFILECORRUPT

  LINEERR_NOMEM LINEERR_INVALPARAM

hwndOwner

Specifies the handle of the parent window in which the function may create any dialog windows that are required during installation.

dwPermanentProviderID

Specifies the service provider's permanent provider ID that must match the <PPID> of the [Provider<PPID>] section to be modified.

This function completes the installation of other pieces required by the service provider after its entries in the [Providers] section of TELEPHON.INI have been made. In particular, it installs the [Provider<PPID>] section whose <PPID> matches the passed *dwPermanentProviderID*, including its *NumLines* and *NumPhones* entries with their appropriate values. If the service provider requires any additional privately-defined entries in this section for proper operation, they must also be installed. A typical way to install this section with its entries is to call **TSPI_providerConfig**.

This function must leave the system in a consistent state. It should run to completion, not allowing the user to abort the installation when it is partly completed. If installation fails, any changes should be “backed out” and an error returned. This may imply pre-scanning to verify that a complete installation is possible, before the installation begins. If installation fails, it is the provider’s responsibility to “back out” what was done.

This function is called only once, during installation of the service provider, until there is a call to **TSPI_providerRemove**. It must be called before any other TSPI-defined function. It is the caller’s responsibility to ensure that the service provider’s *ProviderIDx* entry with its <PPID> value matching the *dwPermanentProviderID* and the service provider’s *ProviderFilenamex* entry have already been made in the [Providers] section before calling this function. The corresponding [Provider<PPID>] section with the *NumLines* and *NumPhones* entries should not already exist. If this function returns an error, the caller must handle it appropriately, for example, by removing the [Providers] section entries if they were being added.

The Telephony Control Panel applet supplied with the Windows Telephony SDK calls this function (with external sequence requirements met as described here) when the “add” command is invoked. It does not call **TSPI_providerConfig** for the “add” command.

The Telephony Control Panel applet will also make the appropriate updates to the [Windows Telephony] section of the WIN.INI file and issue a WM_WININICHANGE message to notify TAPI.DLL of any changes made to this file. TAPI.DLL invokes **TSPI_providerShutdown** and waits until all Telephony operations have shutdown to guarantee that each service provider will read the new configuration the next time it is invoked.

There is no corresponding function at the TAPI level. At that level, applications expect to have service providers already installed. Running applications are informed about dynamic reconfiguration through the LINEDEVSTATE_REINIT or PHONESTATE_REINIT values in the **LINE_LINEDEVSTATE** or **PHONE_STATE** messages.

TSPI_providerRemove

[New - Windows 95]

```
LONG TSPI_providerRemove(HWND hwndOwner, DWORD dwPermanentProviderID)
```

Asks the user to confirm elimination of the service provider and removes the service provider's [Provider<PPID>] section from TELEPHON.INI. It also removes any INI file of its own and entries in other INI files (such as SYSTEM.INI for drivers) it may have. It removes any other modules and files that are no longer needed.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

  LINEERR_OPERATIONFAILED LINEERR_INIFILECORRUPT

  LINEERR_NOMEM LINEERR_INVALPARAM

hwndOwner

Specifies the handle of the parent window in which the function may create any dialog windows required during the removal.

dwPermanentProviderID

Specifies the service provider's permanent provider ID that must match the <PPID> of the [Provider<PPID>] section to be removed.

This function must guarantee that the service provider's section and privately-defined information for the service provider is removed from TELEPHON.INI if it returns success. In particular, the [Provider<PPID>] section whose <PPID> matches *dwPermanentProviderID* must be removed, including its *NumLines* and *NumPhones* entries. If the function returns success, it is the caller's responsibility to remove the matching *ProviderIDx* and *ProviderFilenamex* entries from the [Providers] section, and renumber the remaining entries in the [Providers] section accordingly.

This procedure must leave the system in a consistent state. It should run to completion, not allowing the user to abort the removal when it is partly completed. If removal fails, any changes should be "backed out" and an error returned. This may imply pre-scanning to verify that a complete removal is possible, before the removal begins. If removal fails, it is the provider's responsibility to "back out" what was done.

This function may be called while the service provider is in use (that is, between **TSPI_providerInit** and **TSPI_providerShutdown**). If this happens, the service provider should do an appropriate combination of displaying a user dialog to announce any conflict and confirm removal, restricting removal options to those that can be performed transparently, or issuing **LINE_CLOSE** and **PHONE_CLOSE** messages to inform TAPI.DLL and applications that the affected devices have been forcibly closed for removal. In any case, any changes that affect the behavior visible through the TSPI should take effect only when the service provider is shut down at the next **TSPI_providerShutdown**.

Note that this function should not return LINEERR_INUSE or other errors that might occur because the provider is in use by an application; instead, the provider should confer with the user directly about this problem, and then return LINEERR_OPERATIONFAILED if the user decides to abort the operation.

This procedure is called only once, at the time of removal of the service provider, until there is a call to **TSPI_providerInstall**. It is the responsibility of the caller to ensure this.

The Telephony Control Panel applet supplied with the Windows Telephony SDK calls this function (with external sequence requirements met as described here) when the “remove” command is invoked.

The Telephony Control Panel applet will also make the appropriate updates to the [Windows Telephony] section of the WIN.INI file and issue a WM_WININICHANGE message to notify TAPI.DLL of any changes made to this file. TAPI.DLL invokes **TSPI_providerShutdown** and waits until all Telephony operations have shutdown to guarantee that each service provider will read the new configuration the next time it is invoked.

There is no corresponding function at the TAPI level. At that level, applications expect to have service providers already installed; otherwise their lines and phones do not appear within the available sequence of device IDs. Running applications are informed about dynamic reconfiguration, including removal of service providers, through the LINEDEVSTATE_REINIT or PHONESTATE_REINIT values in the **LINE_LINEDEVSTATE** or **PHONE_STATE** messages.

TSPI_providerShutdown

[New - Windows 95]

```
LONG TSPI_providerShutdown(DWORD dwTSPIVersion)
```

Shuts down the service provider. The service provider should terminate any activities it has in progress and release any resources it has allocated. In addition, it must discard or invalidate any information it has acquired from reading TELEPHON.INI earlier, since this may have changed by the time of the next call to **TSPI_providerInit**.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

  LINEERR_INCOMPATIBLEAPIVERSION

  LINEERR_NOMEM

  LINEERR_OPERATIONFAILED

dwTSPIVersion

Specifies the version of the TSPI definition under which this function must operate. The caller may use **TSPI_lineNegotiateTSPIVersion** or **TSPI_phoneNegotiateTSPIVersion** with the special *dwDeviceID* INITIALIZE_NEGOTIATION to negotiate a version that is guaranteed to be acceptible to the service provider.

It is the caller's responsibility to ensure that the call to this function is strictly paired with a preceeding call to **TSPI_providerInit**. These pairs may overlap, for example, when the Telephony Control Panel applet is used while Telephony operations are in progress. The call to this function must be ignored (returning success) if an outstanding pair remains.

The final paired call to this function must be the last call to any of the TSPI functions prefixed with **TSPI_line** or **TSPI_phone** other than **TSPI_lineNegotiateTSPIVersion,** or **TSPI_phoneNegotiateTSPIVersion**. It is the caller's responsibility to ensure this.

This function should always succeed except in extraordinary circumstances. Most callers will probably ignore the return code since they will be unable to compensate for any error that occurs. The specified return values are more advisory for development diagnostic purposes than anything else.

There is no directly corresponding function in TAPI. In TAPI, multiple different usage instances can be outstanding, with an “application handle” parameter to identify the instance to be operated on. In TSPI, the interface architecture supports only a single usage instance for each distinct service provider.

Constants

LINERR_ Constants

LINERR_ Constants

LINEERR_INCOMPATIBLEAPIVERSION

The passed TSPI version did not match an interface version definition supported by the service provider.

LINEERR_INIFILECORRUPT

A required INI file was corrupt in a way that the removal procedure was unable to correct or handle.

LINEERR_INVALPARAM

A parameter (such as the window handle) was invalid.

LINEERR_NOMEM

Unable to allocate or lock memory.

LINEERR_NOMULTIPLEINSTANCE

TSPI_providerInit was called more than once without an intervening **TSPI_providerShutdown** (that is, an attempt was made to invoke multiple instances of the provider), when the provider does not support multiple instances.

LINEERR_OPERATIONFAILED

The installation or configuration failed or was cancelled by the user. Alternately, the operation failed for an unknown or unspecified reason.

LINEERR_RESOURCEUNAVAIL

The service provider does not have enough resources available to complete the request.

TSPI Line Functions

TSPI Line Functions

TSPI Line Functions

About Line Functions

Introduction

This chapter contains an alphabetical list of the available functions in the Telephony SPI. The information for each function includes the following:

- The correct syntax for the function.
- A statement of the function's purpose.
- A description of the functions's parameters.
- A description of the function's return value.
- A list of the valid call states on entry of the function and typical call state transitions when the request completes. Note that the actual states in which a function may be performed may be further limited by the capabilities of the service provider; service providers must set the dwCallFeatures field in the LINECALLSTATUS structure, the dwAddressFeatures field in the LINEADDRESSSTATUS structure, and the dwLineFeatures field in the LINEDEVSTATUS structure to indicate to applications whether or not a function is permitted at that point in time.
- Comments on using the function and related functions.
- References to other functions, messages, or data structures.
- A description of which fields of large data structures must be filled in by the service provider and which fields must have their values preserved intact.
- A comparison with a corresponding function within TAPI.

Synchronous versus Asynchronous Requests

Some TSPI functions operate synchronously, while others operate asynchronously. Synchronous functions execute the entire request before the caller's thread is allowed to return from the function call. Asynchronous functions return from the function call before the request has executed in its entirety. When the asynchronous request later completes, the service provider reports the completion by calling a callback procedure originally supplied to it by TAPI.DLL early in the initialization sequence.

To support asynchronous functions, TAPI.DLL passes an asynchronous request ID to the service provider as part of the function call. Request IDs are positive integers strictly greater than zero. The service provider does whatever is required to start the operation and then returns the request ID to TAPI.DLL. When the asychronous operation eventually completes, the service provider reports completion by calling a special callback procedure (ASYNC_COMPLETION) in TAPI.DLL. The parameters to this procedure are the original request ID and the result code. Result values are zero for success and negative for error indications.

For synchronous functions, the service provider simply runs the entire operation to completion. If the operation is successful, it returns zero. If an error occurs, it returns a negative error number. TAPI does not pass a request ID to synchronous functions.

It is interesting to consider the timing of a "completion" callback relative to when the original request returns. A typical asynchronous request would be implemented by the service provider as in the following pseudo-code:

```
Some_request(Request_ID, ...) {
  check parameters for validity
  check device state for validity
  store Request_ID for Completion Interrupt Handler's use
  manipulate device registers to start physical operation
  return Request_ID  // to indicate asynch operation
}
Operation Completion Interrupt Handler: {
  if operation was successful then
    call "completion" callback passing Request_ID and "success"
  else
    call "completion" callback passing Request_ID and "error num"
  endif
}
```

If the operation completes very rapidly (or the original request returns very slowly), it is possible that the Interrupt Handler that runs when a physical operation completes may be triggered before the original request returns to the application or even TAPI.DLL. This behavior is allowed. TAPI.DLL guarantees that the eventual completion notification to the application will be delivered after the original request returns.

Synchronous Requests

Service provider designers should try to keep the execution time of synchronous operations short. For example, there is an “Open” operation called at initialization time that prepares the service provider and TAPI DLL for subsequent operations on a device. A service provider whose implementation is split between the client computer and a dedicated server may have reasonable confidence that it can communicate with its remote server. Its implementation of “Open” might just allocate and initialize data structures to represent the device and return, postponing end-to-end communication until some real operation is requested.

Any such “optimistic” implementation of a synchronous operation can later encounter resource limitations or remote communication failures. In general, these cannot be entirely prevented even by a “conservative” approach; service-provider designers will have to choose the best tradeoff between reliability and performance. A failure of this kind should be handled gracefully wherever possible. From the point of view of TSPI devices should remain valid for “bookkeeping” purposes even if they return failure status for any operation requested.

Implementing an optimistic approach for synchronous requests should not be done at the expense of correct semantics for the operation. Returning to the “Open” example, if opening a device needs to compete for and reserve a scarce, non-sharable resources such as communication ports, it should probably do so even if it is time-consuming.

Asynchronous Spontaneous Events

There is another class of asynchronous events besides those described above. These events are “spontaneous” in the sense that they are not direct results of corresponding requests. These events are detected in such cases as when an incoming call arrives, or when an outbound call goes from a “ringing” to an “answering” state. When TAPI.DLL first initializes interactions with the TSPI for a particular device, it passes a pointer to a callback procedure to be called for reporting such spontaneous events. Along with this procedure pointer is a device handle that the service provider includes as one of the actual parameters to the callback.

Memory Allocation

Several TSPI functions return large amounts of information. Applications are responsible for allocating memory for this information; the service provider simply fills in the data. Some of these data structure fields are filled in by TAPI.DLL, and others are filled in by the service provider. The function descriptions in this chapter explain the division of responsibility between TAPI.DLL and the service provider. The service provider must preserve the values in fields for which TAPI.DLL is responsible. In any case, the service provider must fill in the fields it “owns” before reporting the operation complete. Note that, for functions that complete asynchronously, the filling of the data structure may be done synchronously or asynchronously, which in any case must be completed before the **LINE_REPLY** message is sent. When passed to the service provider from TAPI, the structure is filled with zeros, and then the TAPI fields will be added, so it is not necessary for the service provider to explicitly write zero values to any field.

Call States

Most functions on calls only make sense while the call is in one of certain call states. This call state is checked, and the function returns an error if the call is not in one these states. The effect of the successful execution of a request may be some typical call state transitions. The service provider notifies TAPI DLL about call state changes using call state messages. These messages report the state the call has just entered to TAPI DLL and to applications. Since unsolicited events may occur to calls (as when the other party disconnects), TAPI.DLL cannot assume that requests it issues will always result in certain fixed call state transitions.

This model gives the service provider designer maximum flexibility across different telephony environments. For example, it allows designs in which requests propagate through a network to a telelphony server such as a PBX while call state change reports propagate back to the local client at some later time. Such a design does not need to try to maintain a lock-step “view” of the current state of the call in both client and server that is always identical between the two sites.

Overall comparison with TAPI

The TSPI specification is very closely related to the TAPI specification. Most of the functions in one interface have a corresponding function in the other interface. The usual correspondence is as follows:

- TSPI functions have the same names as TAPI functions except that they are prepended with "**TSPI_**". For example, the TAPI function **lineAccept** corresponds to the TSPI function **TSPI_lineAccept**.
- Procedures that allow asynchronous operation insert a *dwRequestID* parameter of type DRV_REQUESTID as their first parameter. This parameter specifies the value to be returned to indicate asynchronous operation and to report asynchronous completion.
- *hCall* parameters of type HCALL are replaced with *hdCall* parameters of type HDRVCALL, indicating the service provider's handle for the call.
- *hLine* parameters of type HLINE are replaced with *hdLine* parameters of type HDRVLINE, indicating the service provider's handle for the line.
- *hPhone* parameters of type HPHONE are replaced with *hdPhone* parameters of type HDRVPHONE, indicating the service provider's handle for the phone.
- Procedures that create a new call, such as **TSPI_lineMakeCall**, replace a single *lphCall* parameter with two parameters: a *htCall* of type HTAPICALL that passes in TAPI.DLL's handle for the call, and a *lphdCall* of type LPHDRVCALL that indicates the location to which the service provider must write its new handle for the call. A similar split of parameters occurs in **TSPI_lineOpen** and **TSPI_phoneOpen**.
- At the TSPI level, there is no notion of "privilege" associated with device handles. Furthermore, at the API level, each application that has a device or call handle has a different handle, but TAPI merges these into a single handle in the service provider side. As a consequence, the error codes and status messages relating to privilege and number of clients using a device do not appear at the TSPI level.

Error Checking

At the TAPI level, an application can pass a variety of different parameters, many of which may be invalid. TAPI.DLL checks for bad parameters and returns errors to the application without calling the service provider. Each function description at the TSPI level describes the parameter errors that have already been tested. The service provider need not repeat these tests, though it must perform any additional validity tests appropriate to the function. Titles and descriptions of common parameter validity tests that appear in many functions are listed below.

pointer validity TAPI.DLL has already tested pointers to data storage to make sure that they point to readable or writeable memory of the size appropriate to the operation. In addition, for variably sized data structures starting with a **dwTotalSize** field, the data structure has been verified to ensure that the indicated total size is really available.

fixed size validity For variably sized data strucutres, the data structure has been checked to make sure that there is at least enough space for the fixed size part of the data structure and that **dwTotalSize** is sufficient to cover the fixed part.

offset/size zeroed For variably sized data structures, the "...**Offset**" and "...**Size**" fields corresponding to parts that the service provider fills in have been preset with zero values before the service provider was called.

handle validity TAPI.DLL guarantees that line, phone, and call handles (of defined types HDRVLINE, HDRVPHONE, and HDRVCALL) are valid. That is, they are values that have been returned without error as handles in **TSPI_lineOpen**, **TSPI_phoneOpen**, or one of the following that starts the lifetime of a call handle:

TSPI_lineMakeCall

TSPI_lineCompleteTransfer

TSPI_lineForward

TSPI_linePickup

TSPI_linePrepareAddToConference

TSPI_lineSetupConference

TSPI_lineSetupTransfer

TSPI_lineUnpark

LINE_NEWCALL messages

Reference

Callback Functions

- **ASYNC_COMPLETION**
- **LINEEVENT**
- **PHONEEVENT**

ASYNC_COMPLETION

[New - Windows 95]

```
ASYNC_COMPLETION Completion_Proc;
void (CALLBACK* Completion_Proc) (DRV_REQUESTID dwRequestID,
    LONG lResult)
```

This type is a placeholder for a callback function implemented by TAPI.DLL and supplied to the service provider early in the initialization sequence. The service provider calls this function to report the completion of a line or phone procedure that it executes asynchronously.

- No return value.

dwRequestID

Specifies the ID that was passed in the original request that the service provider executed asynchronously.

lResult

Specifies the outcome of the operation. This can be zero to indicate success or a negative number to indicate an error. The possible specific error values that may result from a function are the same for asynchronous or synchronous execution.

The call state when calling this function can be any state.

This procedure is supplied by TAPI.DLL at the time a service provider is initialized with the **TSPI_providerInit** function. Some of the TSPI procedures that operate on line, call, and phone devices specify asynchronous operation. These procedures include a *dwRequestID* parameter to identify the request. When such a procedure is called, the service provider may return a negative number for an error if one is detected immediately, or the positive *dwRequestID* if the operation continues asynchronously. The service provider must report completion exactly once for each request it executes asynchronously. It does so by calling this procedure. The service provider is not permitted to call this procedure or the **LINEEVENT** or **PHONEEVENT** procedure again until this procedure returns.

The service provider is permitted to call the **ASYNC_COMPLETION** procedure before it returns from the original request. **ASYNC_COMPLETION** may be called from within an interrupt context. TAPI.DLL guarantees not to call the service provider from within the **ASYNC_COMPLETION** context except where noted.

This does not have any direct correspondence at the TAPI level since at that level asynchronous function completions are reported as a message passed through the same callback interface that is used for spontaneous event messages. At the TSPI level, spontaneous events are reported through the **LINEEVENT** and **PHONEEVENT** callback procedures.

LINEEVENT

[New - Windows 95]

```
LINEEVENT  Line_Event;
void (CALLBACK * Line_Event) (HTAPILINE htLine, HTAPICALL htCall,
    DWORD dwMsg, DWORD dwParam1, DWORD dwParam2, DWORD dwParam3)
```

This type is a placeholder for a callback function implemented by TAPI.DLL and supplied to the service provider as a parameter to **TSPI_lineOpen**. The service provider calls this function to report spontaneously occurring events on the line or on calls on the line.

- No return value.

htLine

Specifies TAPI.DLL's handle for the line on which the event occurred.

htCall

Specifies TAPI.DLL's handle for the call on which the event occurred if this is a call-related event. For line-related events where there is no call, this parameter should be set to zero.

dwMsg

Specifies what kind of event is being reported. Interpretation of the other parameters is done in different ways according to the context indicated by *dwMsg*. The values of this parameter are defined in TAPI.H, and correspond to the names of the messages themselves, for example, LINE_LINEDEVSTATE.

dwParam1

Specifies a parameter for the message.

dwParam2

Specifies a parameter for the message.

dwParam3

Specifies a parameter for the message.

The call state when calling this function can be any state.

The service provider passes the **HTAPILINE** value supplied to **TSPI_lineOpen** as the *htLine* parameter. It includes the message identifier and parameters specific to the event.

The **LINEEVENT** procedure may be called from within an interrupt context.

This function differs from the callback function defined at the TAPI level in that it separates "line" and "call" parameters. Both parameters are used for some messages. The sets of messages that can be passed to this procedure differs slightly from the TAPI level. In particular, completion of asynchronously executing requests is reported through the **ASYNC_COMPLETION** callback instead of this one.

PHONEEVENT

[New - Windows 95]

```
PHONEEVENT  Phone_Event;
void (CALLBACK * Phone_Event) (HTAPIPHONE htPhone, DWORD dwMsg,
    DWORD dwParam1, DWORD dwParam2, DWORD dwParam3)
```

This type is a placeholder for a callback function implemented by TAPI.DLL and supplied to the service provider at the time a phone device is opened. The service provider calls this function to report spontaneously occurring events on the phone.

- No return value.

htPhone

Specifies TAPI.DLL's handle for the phone on which the event occurred.

dwMsg

Specifies what kind of event is being reported. Interpretation of the other parameters is done in different ways according to the context indicated by *dwMsg*. The values of this parameter are defined in TAPI.H, and correspond to the names of the messages themselves, for example, **LINE_LINEDEVSTATE**.

dwParam1

Specifies a parameter for the message.

dwParam2

Specifies a parameter for the message.

dwParam3

Specifies a parameter for the message.

The call state when calling this function can be any state.

The service provider passes the **HTAPIPHONE** value supplied to **TSPI_phoneOpen** as the *htPhone* parameter. It includes the message identifier and parameters specific to the event.

The **PHONEEVENT** procedure may be called from within an interrupt context.

The sets of messages that can be passed to this procedure differs slightly from the messages to the corresponding callback at the TAPI level. In particular, completion of asynchronously executing requests is reported through the **ASYNC_COMPLETION** callback instead of this one.

Functions

- TSPI_lineAccept
- TSPI_lineAddToConference
- TSPI_lineAnswer
- TSPI_lineBlindTransfer
- TSPI_lineClose
- TSPI_lineCloseCall
- TSPI_lineCompleteCall
- TSPI_lineCompleteTransfer
- TSPI_lineConditionalMediaDetection
- TSPI_lineConfigDialog
- TSPI_lineDevSpecific
- TSPI_lineDevSpecificFeature
- TSPI_lineDial
- TSPI_lineDrop
- TSPI_lineForward
- TSPI_lineGatherDigits
- TSPI_lineGenerateDigits
- TSPI_lineGenerateTone
- TSPI_lineGetAddressCaps
- TSPI_lineGetAddressID
- TSPI_lineGetAddressStatus
- TSPI_lineGetCallAddressID
- TSPI_lineGetCallInfo
- TSPI_lineGetCallStatus
- TSPI_lineGetDevCaps
- TSPI_lineGetDevConfig
- TSPI_lineGetExtensionID
- TSPI_lineGetIcon
- TSPI_lineGetID
- TSPI_lineGetLineDevStatus
- TSPI_lineGetNumAddressIDs
- TSPI_lineHold
- TSPI_lineMakeCall
- TSPI_lineMonitorDigits
- TSPI_lineMonitorMedia
- TSPI_lineMonitorTones
- TSPI_lineNegotiateExtVersion
- TSPI_lineNegotiateTSPIVersion
- TSPI_lineOpen
- TSPI_linePark
- TSPI_linePickup
- TSPI_linePrepareAddToConference
- TSPI_lineRedirect
- TSPI_lineRemoveFromConference

TSPI_lineAccept

[New - Windows 95]

```
LONG TSPI_lineAccept(DRV_REQUESTID dwRequestID, HDRVCALL hdCall,
    LPCSTR lpsUserUserInfo, DWORD dwSize)
```

Accepts the specified offered call. It may optionally send the specified user-to-user information to the calling party.

- Returns *dwRequestID* if the function will be completed asynchronously, or a negative error number if an error has occurred. The *lResult* parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_OPERATIONFAILED
LINEERR_INVALCALLSTATE	LINEERR_RESOURCEUNAVAIL
LINEERR_NOMEM	LINEERR_USERUSERINFOTOOBIG
LINEERR_OPERATIONUNAVAIL	

dwRequestID

Specifies the identifier of the asynchronous request.

hdCall

Specifies the handle to the call to be accepted. The call state of *hdCall* can be *offering*.

lpsUserUserInfo

Specifies a far pointer to a string containing user-to-user information to be sent to the remote party as part of the call accept. This pointer is NULL if no user-to-user information is to be sent. User-to-user information is only sent if supported by the underlying network (see **LINEDEVCAPS**).

dwSize

Specifies the size in bytes of the user-to-user information in *lpsUserUserInfo*. If *lpsUserUserInfo* is NULL, *dwSize* should be ignored.

TSPI_lineAccept is used in telephony environments (such as ISDN) that allow alerting associated with incoming calls to be separate from the initial offering of the call. When a call comes in, the call is first offered. For some small time duration, the client application may have the option to reject the call using **TSPI_lineDrop**, redirect the call to another station using **TSPI_lineRedirect**, answer the call using **TSPI_lineAnswer**, or accept the call using **TSPI_lineAccept**. After a call has been successfully accepted, alerting at both the called and calling device will begin, and the call state will typically transition to the *accepted* state. The service provider must set the flag LINEADDRCAPFLAGS_ACCEPTTOALERT in the **dwAddrCapFlags** field of the **LINEADDRESSCAPS** data structure if the application must call **TSPI_lineAccept** in order for alerting to begin.

To TAPI.DLL, alerting is reported using the **LINE_LINEDEVSTATE** message with the *ringing* indication.

TSPI_lineAccept may also be supported by non-ISDN service providers. The call state transition to the *accepted* state can be used by other of TAPI.DLL's clients as an indication that some application has claimed responsibility for the call and has presented the call to the user.

The client application has the option to send user-to-user information at the time of the accept. Even if user-to-user information can be sent, often no guarantees are made that the network will deliver this information to the calling party. The client application may consult a line's device capabilities to determine whether call accept is available.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

TSPI_lineAddToConference

[New - Windows 95]

```
LONG TSPI_lineAddToConference(DRV_REQUESTID dwRequestID,
    HDRVCALL hdConfCall, HDRVCALL hdConsultCall)
```

Adds the call specified by *hdConsultCall*, to the conference call specified by *hdConfCall*.

- Returns *dwRequestID* or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

  LINEERR_INVALCALLHANDLE LINEERR_OPERATIONUNAVAIL
  LINEERR_INVALCALLSTATE LINEERR_OPERATIONFAILED
  LINEERR_CONFERENCEFULL LINEERR_RESOURCEUNAVAIL
  LINEERR_NOMEM

dwRequestID

Specifies the identifier of the asynchronous request.

hdConfCall

Specifies the handle to the conference call. The call state of *hdConfCall* can be *onHoldPendingConference*, *onHold*.

hdConsultCall

Specifies the handle to the call to be added to the conference call. This call cannot be either a parent of another conference or a participant in any conference. Depending on the device capabilities indicated in **LINEADDRESSCAPS**, the *hdConsultCall* may not necessarily have been established using **TSPI_lineSetupConference** or **TSPI_linePrepareAddToConference**. The call state of *hdConsultCall* can be *connected*, *onHold*.

The servicer provider returns LINEERR_INVALCALLHANDLE if *hdConsultCall* is a parent of another conference or already a participant in a conference, or *hdConsultCall* cannot be added for other reasons, such as it must have been established using **TSPI_lineSetupConference** or **TSPI_linePrepareAddToConference**.

Note that the call handle of the added party remains valid after adding the call to a conference; its state will typically change to *conferenced* while the state of the conference call will typically become *connected.* The handle to an individual participating call can be used later to remove that party from the conference call using **TSPI_lineRemoveFromConference**.

The call states of the calls participating in a conference are not independent. For example, when dropping a conference call, all participating calls may automatically become *idle*. TAPI.DLL may consult the line's device capabilities to determine what form of conference removal is available. TAPI.DLL or its client applications should track the **LINE_CALLSTATE** messages to determine what really happened to the calls involved.

The conference call is established either through **TSPI_lineSetupConference** or **TSPI_lineCompleteTransfer**. The call added to a conference will typically be established using **TSPI_lineSetupConference** or **TSPI_linePrepareAddToConference**. Some switches may allow adding of an arbitrary calls to conference, and such a call may have been set up using **TSPI_lineMakeCall** and be on (hard) hold. All calls that are part of a conference must exist on the same open line.

Any monitoring (media, tones, digits) on a conference call applies only to the *hdConfCall*, not to the individual participating calls.

This function has no restrictions based on privilege as in the corresponding function at the TAPI level. There is no explicit requirement for the service provider to track the relationships between the "parent" conference call and its participants, since there is no TSPI correspondence to the TAPI function **lineGetCalls**. Many service providers may find it necessary to track these relationships internally to implement the other conference call management functions.

TSPI_lineAnswer

[New - Windows 95]

```
LONG TSPI_lineAnswer(DRV_REQUESTID dwRequestID, HDRVCALL hdCall,
    LPCSTR lpsUserUserInfo, DWORD dwSize)
```

Answers the specified offering call.

- Returns *dwRequestID* or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALCALLSTATE	LINEERR_OPERATIONFAILED
LINEERR_INUSE	LINEERR_RESOURCEUNAVAIL
LINEERR_NOMEM	LINEERR_USERUSERINFOTOOBIG

dwRequestID

Specifies the identifier of the asynchronous request.

hdCall

Specifies the service provider's handle to the call to be answered. The call state of *hdCall* can be *offering*, *accepted*.

lpsUserUserInfo

Specifies a far pointer to a string containing user-to-user information to be sent to the remote party at the time of answering the call. If this pointer is NULL, it indicates that no user-to-user information is to be sent. User-to-user information is only sent if supported by the underlying network (as indicated in **LINEDEVCAPS**).

dwSize

Specifies the size in bytes of the user-to-user information in *lpsUserUserInfo*. If *lpsUserUserInfo* is NULL *dwSize* is ignored.

When a new call arrives, the service provider sends TAPI.DLL a **LINE_NEWCALL** message to exchange handles for the call. The service provider follows this with a **LINE_CALLSTATE** message inform TAPI.DLL and its client applications of the call's state. A client application typically answers the call using **TSPI_lineAnswer**. After the call has been successfully answered, the call will typically transition to the *connected* state.

In some telephony environments (like ISDN) where user alerting is separate from call offering, TAPI.DLL and its client applications may have the option to first accept a call prior to answering, or instead to reject or redirect the *offering* call.

If a call is offered at the time another call is already active, the new call is connected to by invoking **TSPI_lineAnswer**. The effect this has on the existing active call depends on the line's device capabilities. The first call may be unaffected, it may automatically be dropped, or it may automatically be placed on hold. The appropriate **LINE_CALLSTATE** messages are used to report state transitions to TAPI.DLL about both calls.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

TSPI_lineBlindTransfer

[New - Windows 95]

```
LONG TSPI_lineBlindTransfer(DRV_REQUESTID dwRequestID, HDRVCALL hdCall,
    LPCSTR lpszDestAddress, DWORD dwCountryCode)
```

Performs a blind or single-step transfer of the specified call to the specified destination address.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

  LINEERR_INVALCALLHANDLE LINEERR_NOMEM
  LINEERR_INVALCALLSTATE LINEERR_OPERATIONUNAVAIL
  LINEERR_INVALADDRESS LINEERR_OPERATIONFAILED
  LINEERR_ADDRESSBLOCKED LINEERR_RESOURCEUNAVAIL
  LINEERR_INVALCOUNTRYCODE

dwRequestID
Specifies the identifier of the asynchronous request.

hdCall
Specifies the service provider's handle to the call to be transferred. The call state of *hdCall* can be *connected*.

lpszDestAddress
Specifies a far pointer to a NULL-terminated string identifying where the call is to be transferred to. The destination address uses the standard dialable number format.

dwCountryCode
Specifies the country code of the destination. This should be used by the implementation to select the call progress protocols for the destination address. If a value of zero is specified, the service provider should use a default. Note that *dwCountryCode* is not validated by TAPI.DLL when this function is called.

The service provider carries out no dialing if it returns LINEERR_INVALADDRESS.

Blind transfer differs from a consultation transfer in that no consultation call is made visible to TAPI.DLL. After the blind transfer successfully completes the specified call will typically be cleared from the line it was on and transition to the *idle* state. Note that the service provider's call handle must remain valid after the transfer has completed. TAPI.DLL causes this handle to be invalidated when it is no longer interested in the transferred call using **TSPI_lineCloseCall**.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

TSPI_lineClose

[New - Windows 95]

```
LONG TSPI_lineClose(HDRVLINE hdLine)
```

Closes the specified open line device after completing or aborting all outstanding calls and asynchronous operations on the device.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

  LINEERR_NOMEM LINEERR_OPERATIONFAILED
  LINEERR_OPERATIONUNAVAIL LINEERR_RESOURCEUNAVAIL

hdLine

Specifies the service provider's handle to the line to be closed. After the line has been successfully closed, this handle is no longer valid.

The service provider has the responsibility to report completion for every asynchronous operation. If **TSPI_lineClose** is called for a line on which there are outstanding asynchronous operations, the operations should be reported complete with an appropriate result or error code before this procedure returns.

A similar requirement exists for active calls on the line. Outstanding operations must be reported complete with appropriate result or error codes. Active calls may also be dropped, if required, and if this behavior was indicated by the LINEDEVCAPFLAGS_CLOSEDROP bit in the **LINEDEVCAPS** structure.

After this procedure returns, the service provider must report no further *htCall* on the line or calls that were on the line. The service provider's handles for the line and calls on the line become "invalid."

The service provider must relinquish non-sharable resources it reserves while the line is open. For example, closing a line accessed through a comm port and modem should result in closing the comm port, making it once again available for use by other applications.

The service provider does not issue the **LINE_LINEDEVSTATE** message in response to this function invocation, since the line becomes closed and there is no longer any interest in its state changes.

TSPI_lineCloseCall

[New - Windows 95]

```
LONG TSPI_lineCloseCall(HDRVCALL hdCall)
```

Deallocates the call after completing or aborting all outstanding asynchronous operations on the call.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_NOMEM	LINEERR_OPERATIONFAILED
LINEERR_OPERATIONUNAVAIL	LINEERR_RESOURCEUNAVAIL

hdCall

Specifies the service provider's handle to the call to be closed. After the call has been successfully closed, this handle is no longer valid. The call state can be any.

The service provider has the responsibility to report completion for asynchronous operations. If **TSPI_lineCloseCall** is called for a call on which there are outstanding asynchronous operations, the operations should be reported complete with an appropriate result or error code before this procedure returns. After this procedure returns, the service provider must report no further events on the call. The service provider's handles for the line and calls on the line become "invalid."

If there is an active call on the line at the time of **TSPI_lineCloseCall**, the call may be dropped if this behavior had been indicated by the LINEDEVCAPFLAGS_CLOSEDROP bit in the **LINEDEVCAPS** structure.

If the service provider can determine that there is another agent sharing control of the call, such as in a "party line" situation with a separate handset, the service provider should simply let control of the call pass to the other agent rather than forcibly dropping it.

This function should always succeed except in extraordinary circumstances. Most callers will probably ignore the return code since they will be unable to compensate for any error that occurs. The specified return values are more advisory for development diagnostic purposes than anything else.

This function is called when the last application with a handle to this call executes **lineDeallocateCall**.

TSPI_lineCompleteCall

[New - Windows 95]

```
LONG TSPI_lineCompleteCall(DRV_REQUESTID dwRequestID, HDRVCALL hdCall,
    LPDWORD lpdwCompletionID, DWORD dwCompletionMode, DWORD dwMessageID)
```

Is used to specify how a call that could not be connected normally should be completed instead. The network or switch may not be able to complete a call because network resources are busy or the remote station is busy or doesn't answer.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_NOMEM
LINEERR_INVALCALLSTATE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALCALLCOMPLMODE	LINEERR_OPERATIONFAILED
LINEERR_INVALPOINTER	LINEERR_RESOURCEUNAVAIL
LINEERR_COMPLETIONOVERRUN	LINEERR_INVALMESSAGEID

dwRequestID

Specifies the identifier of the asynchronous request.

hdCall

Specifies the service provider's handle to the call whose completion is requested. The call state of *hdCall* can be *busy*, *ringback*.

lpdwCompletionID

Specifies a far pointer to a DWORD-sized memory location where the service provider writes a completion ID. This uniquely identifies a completion request in progress on the line that contains the *hdCall*. In particular, a completion ID becomes invalid after the request completes or is cancelled using the function **TSPI_lineUncompleteCall**. The service provider is free to reuse the completion ID as soon as it becomes invalid.

dwCompletionMode

Specifies the way in which the call is to be completed. This parameter uses the following **LINECALLCOMPLMODE_** constants. Only one of the indicated flags may be set at a time.

LINECALLCOMPLMODE_CAMPON
: Queues the call until the call can be completed. The call remains in the *busy* state while queued.

LINECALLCOMPLMODE_CALLBACK
: Requests the called station to return the call when the called station becomes idle. The call typically transitions to the *idle* state.

LINECALLCOMPLMODE_INTRUDE
: Requests that the call be added to an existing physical call at the called station (barge in). The call typically transitions to the *connected* state when the intrusion occurs.

LINECALLCOMPLMODE_MESSAGE
: Leave a short predefined message for the called station (Leave Word Calling). The message to be sent is specified in *dwMessageID*. The call typically transitions to the *idle* state.

dwMessageID

Specifies the message that is to be sent when completing the call using LINECALLCOMPLMODE_MESSAGE. This ID selects the message from a small number of predefined messages. Note that this parameter is not validated by TAPI.DLL when this function is called.

This function is considered complete when the request has been accepted by the network or switch; not when the request is fully completed in the way specified. When the called station or network enters a state where the call can be completed as requested, the service privider must send a **LINE_CALLSTATE** message with the call state equal to *offering*. The call's **LINECALLINFO** record will list the reason for the call as CALLCOMPLETION and provide the completion ID as well. It is possible

to have multiple call completion requests outstanding at any given time; the maximum number is device dependent. The completion ID is also used to refer to each individual request so requests can be canceled by calling **TSPI_lineUncompleteCall**.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

TSPI_lineCompleteTransfer

[New - Windows 95]

```
LONG TSPI_lineCompleteTransfer(DRV_REQUESTID dwRequestID,
    HDRVCALL hdCall, HDRVCALL hdConsultCall, HTAPICALL htConfCall,
    LPHDRVCALL lphdConfCall, DWORD dwTransferMode)
```

Completes the transfer of the specified call to the party connected in the consultation call. If *dwTransferMode* is LINETRANSFERMODE_CONFERENCE, the original call handle is changed to a conference call. Otherwise, the service provider should send call state messages changing the calls to *idle*.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALCALLSTATE	LINEERR_OPERATIONFAILED
LINEERR_NOMEM	LINEERR_RESOURCEUNAVAIL

dwRequestID

Specifies the identifier of the asynchronous request.

hdCall

Specifies the service provider's handle to the call to be transferred. The call state of *hdCall* can be *onHoldPendingTransfer*.

hdConsultCall

Specifies a handle to the call that represents a connection to the destination of the transfer. The call state of *hdConsultCall* can be *connected*, *ringback*, *busy*.

htConfCall

This parameter is only valid if *dwTransferMode* is specified as LINETRANSFERMODE_CONFERENCE. It should be used to replace the *htCall* associated with the original *hdCall*. The service provider must save this parameter value and use it in all subsequent calls to the **LINEEVENT** procedure reporting events on the call. Otherwise this parameter is ignored.

lphdConfCall

Specifies a far pointer to an HDRVCALL representing the service provider's identifier for the call. This parameter is only valid if *dwTransferMode* is specified as LINETRANSFERMODE_CONFERENCE. The service provider must fill this location with its handle for the new conference call.

dwTransferMode

Specifies how the initiated transfer request is to be resolved. This parameter uses the following LINETRANSFERMODE_ constants:

LINETRANSFERMODE_TRANSFER
: Resolve the initiated transfer by transferring the initial call to the consultation call.

LINETRANSFERMODE_CONFERENCE
: Resolve the initiated transfer by conferencing all three parties into a three-way conference call.

This function completes the transfer of the original call, *hdCall*, to the party currently connected through *hdConsultCall*. The consultation call will typically have been dialed on the consultation call allocated as part of **TSPI_lineSetupTransfer**, but it may be any call to which the switch is capable of transferring *hdCall*.

The transfer request can be resolved either as a transfer or as a three-way conference call. When resolved as a transfer, the parties connected through *hdCall* and *hdConsultCall* will be connected to each other, and both *hdCall* and *hdConsultCall* transition to the idle state.

When resolved as a conference, all three parties will enter into a conference call. Both existing call handles remain valid, but will transition to the conferenced state. A conference call handle will be created and returned, and it will transition to the *connected* state.

It may also be possible to perform a blind transfer of a call using **TSPI_lineBlindTransfer**.

This function differs from the corresponding TAPI function in that it follows the TSPI model for beginning the lifetime of a call. TAPI.DLL and the service provider exchange opaque handles representing the call with one another. In addition, the service provider is permitted to do callbacks for the new call before it returns from this procedure. In any case, the service provider must also treat the handle it returned as “not yet valid” until after the matching **ASYNC_COMPLETION** message reports success. In other words, it must not issue any **LINEEVENT** messages for the new call or include it in call counts in messages or status data structures for the line.

TSPI_lineConditionalMediaDetection

[New - Windows 95]

```
LONG TSPI_lineConditionalMediaDetection(HDRVLINE hdLine,
    DWORD dwMediaModes, LPLINECALLPARAMS const lpCallParams)
```

Is invoked by TAPI.DLL whenever a client application uses LINEMAPPER as the *dwDeviceID* in a **lineOpen** function to request that lines be scanned to find one that supports the desired media modes and call parameters. TAPI.DLL scans based on the union of the desired media mode and the other media modes currently being monitored on the line, to give the service provider the opportunity to indicate if it cannot simultaneously monitor for all of the requested media modes. If the service provider can monitor for the indicated set of media modes AND support the capabilities indicated in *lpCallParams*, it replies with a “success” indication. It leaves the active media monitoring modes for the line unchanged.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

  LINEERR_INVALLINEHANDLE LINEERR_OPERATIONFAILED
  LINEERR_NODRIVER LINEERR_RESOURCEUNAVAIL
  LINEERR_NOMEM LINEERR_INVALMEDIAMODE
  LINEERR_OPERATIONUNAVAIL

hdLine

Specifies the service provider’s handle to the line on which media monitoring and parameter capabilities are to be set.

dwMediaModes

Specifies the media mode(s) currently of interest to the calling application. This parameter uses the following LINEMEDIAMODE_ constants:

LINEMEDIAMODE_UNKNOWN
Unclassified calls.

LINEMEDIAMODE_INTERACTIVEVOICE
Interactive voice media type. That is, it manages live voice calls.

LINEMEDIAMODE_AUTOMATEDVOICE
Voice data managed by an automated application. In cases where the service provider does not distinguish between automated or interactive voice, this should be treated the same as interactive voice.

LINEMEDIAMODE_DATAMODEM
Calls with the data modem media mode.

LINEMEDIAMODE_G3FAX
Calls of the group 3 fax media type.

LINEMEDIAMODE_TDD
Calls with the TDD (Telephony Devices for the Deaf) media mode.

LINEMEDIAMODE_G4FAX
Calls of the group 4 fax media type.

LINEMEDIAMODE_DIGITALDATA
Calls of the digital data media type.

LINEMEDIAMODE_TELETEX
Calls with the teletex media mode.

LINEMEDIAMODE_VIDEOTEX
Calls with the videotex media mode.

LINEMEDIAMODE_TELEX
Calls with the telex media mode.

LINEMEDIAMODE_MIXED
Calls with the ISDN mixed media mode.

LINEMEDIAMODE_ADSI
Calls with the ADSI (Analog Display Services Interface) media mode.

lpCallParams

Specifies a far pointer to a structure of type **LINECALLPARAMS**. It describes the call parameters that the line device should be able to provide. The only relevant fields of *lpCallParams* for the purposes of this test are the following:

DWORD **dwBearerMode***;*
DWORD **dwMinRate**;
DWORD **dwMaxRate**;
DWORD **dwMediaMode**;
DWORD **dwCallParamFlags**;
DWORD **dwAddressMode**;

If **dwAddressMode** is LINEADDRESSMODE_ADDRESSID, it means that any address on the line is acceptable; otherwise if **dwAddressMode** is LINEADDRESSMODE_DIALABLEADDR, indicating that a specific originating address (phone number) is searched for, or if it is a provider-specific extension, **dwOrigAddressSize/Offset** and the portion of the variable part they refer to are also relevant. If **dwAddressMode** is a provider-specific extension, additional information may be contained in the **dwDeviceSpecific** variably sized field. All other fields are irrelevant to the function.

A TAPI **lineOpen** function that specifies a device ID of LINEMAPPER typically results in calling this procedure for multiple line devices to hunt for a suitable line, possibly also opening as-yet unopened lines. A “success” result indicates that the line is suitable for the calling application’s requirements. Note that the media monitoring modes demanded at the TSPI level are the union of monitoring modes demanded by multiple applications at the TAPI level. As a consequence of this, it is most common for multiple media mode flags to be set simultaneously at this level. The service provider must test to determine if it can support at least the specified set regardless of what modes are currently in effect. TAPI.DLL insures that the *dwMediaModes* parameter has at least one bit set and that no reserved bits are set. It is the service provider’s responsibility to do any further validity checks on the media modes, such as checking whether any regular or extended media modes are supported by the service provider.

Comparison

There is no directly corresponding function at the TAPI level. This procedure corresponds to the “test” implied for each individual line by the **lineOpen** procedure when it is called with the device ID LINEMAPPER.

TSPI_lineConfigDialog

[New - Windows 95]

```
LONG TSPI_lineConfigDialog(DWORD dwDeviceID, HWND hwndOwner,
    LPCSTR lpszDeviceClass)
```

Causes the provider of the specified line device to display a modal dialog as a child window of *hwndOwner* to allow the user to configure parameters related to the line device.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INUSE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALDEVICECLASS	LINEERR_OPERATIONFAILED
LINEERR_NOMEM	LINEERR_RESOURCEUNAVAIL

dwDeviceID

Specifies the line device to be configured.

hwndOwner

Specifies a handle to a parent window in which the dialog window is to be placed.

lpszDeviceClass

Specifies a far pointer to a NULL-terminated string that identifies a device class name. This device class allows the caller to select a specific subscreen of configuration information applicable to that device class. If this parameter is NULL or an empty string, the highest level configuration dialog should be selected. The permitted strings are the same as for **TSPI_lineGetID**. For example, if the line supports the Comm API, passing “COMM” as *lpszDeviceClass* causes the provider to display the parameters related specifically to Comm (or, at least, to start at the corresponding point in a multilevel configuration dialog chain, so that the user doesn’t have to search to find the desired parameters.)

The procedure must update the [Windows Telephony] section in WIN.INI and broadcast the WM_WININICHANGE message if it makes any changes to TELEPHON.INI that would cause a change in the line or address capabilities reported in **LINEDEVCAPS** or **LINEADDRESSCAPS**, or if a line device is created or removed.

There is no restriction that this function (**TSPI_lineConfigDialog**) be called only when the line is closed. However, each provider can impose such a restriction itself. When **TSPI_lineConfigDialog** is called, the provider could alert the user with the message “The line is in use by one or more applications. You may not change the line configuration while the line is in use” (and return the error message LINEERR_INUSE). However, some configuration may be safe to change “on the fly,” particularly those related to media modes (such as the modem error control protocol), especially when that media mode is not currently in use. The provider could allow those options to be changed while the line is open.

Users should not be allowed to change anything that alters values returned with **LINEDEVCAPS** or **LINEADDRESSCAPS** without first forcibly closing the line as a signal that applications must call functions that return these structures in order to have accurate information.

TSPI_lineDevSpecific

[New - Windows 95]

```
LONG TSPI_lineDevSpecific(DRV_REQUESTID dwRequestID, HDRVLINE hdLine,
    DWORD dwAddressID, HDRVCALL hdCall, LPVOID lpParams, DWORD dwSize)
```

Is used as a general extension mechanism to enable service providers to provide access to features not described in other operations. The meaning of the extensions are device-specific, and taking advantage of these extensions requires the application to be fully aware of them.

- Returns *dwRequestID* or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALLINEHANDLE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALADDRESSID	LINEERR_OPERATIONFAILED
LINEERR_NOMEM	LINEERR_RESOURCEUNAVAIL

dwRequestID

Specifies the identifier of the asynchronous request.

hdLine

Specifies the service provider's handle to the line to be operated on.

dwAddressID

Specifies the address on the specified line to be operated on.

hdCall

Specifies the service provider's handle to the call to be operated on. This field may have the value NULL. The call state of *hdCall* is device specific.

lpParams

Specifies a far pointer to a memory area used to hold a parameter block. The format of this parameter block is device specific.

dwSize

The size in bytes of the parameter block area.

Additional return values are device specific.

This operation provides a generic parameter profile. The interpretation of the parameter structure is device specific. The *hdLine* parameter is always specified by TAPI.DLL. Whether *dwAddressID* and/or *hdCall* are expected to be valid is device-specific. If specified, they must belong to *hdLine*. Indications and replies sent back to the application that are device specific should use the **LINE_DEVSPECIFIC** message.

This function is called in direct response to an application calling the TAPI function **lineDevSpecific**. TAPI.DLL translates the *hLine* and *hCall* parameters used at the TAPI level to the corresponding *hdLine* and *hdCall* parameters used at the TSPI level. The *lpParams* buffer is passed unmodified.

Note also that the *lpParams* data structure should not contain any pointers since they would not be properly translated (thunked) when running a 16-bit application in a 32-bit version of TAPI.DLL and vice versa.

A service provider can provide access to device-specific functions by defining parameters for use with this operation. Applications that want to make use of these device-specific extensions should consult the device-specific (in this case meaning vendor specific) documentation that describes which extensions are defined. An application that relies on these device-specific extensions will typically not be portable to work with other service provider environments.

This operation is part of the extended telephony services. It only provides access to a device-specific feature without defining its meaning. This operation is only available if the application has successfully negotiated and selected a device-specific extension version.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original

thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

TSPI_lineDevSpecificFeature

[New - Windows 95]

```
LONG TSPI_lineDevSpecificFeature(DRV_REQUESTID dwRequestID,
    HDRVLINE hdLine, DWORD dwFeature, LPVOID lpParams, DWORD dwSize)
```

Is used as an extension mechanism to enable service providers to provide access to features not described in other operations. The meaning of these extensions are device-specific, and taking advantage of these extensions requires TAPI.DLL or its client application to be fully aware of them.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALFEATURE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALLINEHANDLE	LINEERR_OPERATIONFAILED
LINEERR_NOMEM	LINEERR_RESOURCEUNAVAIL

dwRequestID

Specifies the identifier of the asynchronous request.

hdLine

Specifies the service provider's handle to the line device.

dwFeature

Specifies the feature to invoke on the line device. This parameter uses PHONEBUTTONFUNCTION_ constants.

lpParams

Specifies a far pointer to a memory area used to hold a feature-dependent parameter block. The format of this parameter block is device specific.

dwSize

Specifies the size of the buffer in bytes.

Additional return values are device specific.

The call state of *hdCall* is device-specific.

This function provides TAPI applications with phone feature button emulation capabilities. When TAPI.DLL invokes this operation on behalf of a client application, it specifies the equivalent of a "button press" event. This method of invoking features is highly device dependent, as the API does not define their meaning. Note that an application that relies on these device-specific extensions will typically not be portable to work with other service provider environments.

This function is called in direct response to an application calling the TAPI function **lineDevSpecificFeature**. TAPI.DLL translates the *hLine* parameter used at the TAPI level to the corresponding *hdLine* parameter used at the TSPI level. The *lpParams* buffer is passed through unmodified.

Note also that the *lpParams* data structure should not contain any pointers since they would not be properly translated (thunked) when running a 16-bit application in a 32-bit version of TAPI.DLL and vice versa.

This operation is part of the extended telephony services. It only provides access to a device-specific feature without defining its meaning. This operation is only available if TAPI.DLL has successfully negotiated and selected a device-specific extension version.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

TSPI_lineDial

[New - Windows 95]

```
LONG TSPI_lineDial(DRV_REQUESTID dwRequestID, HDRVCALL hdCall,
    LPCSTR lpszDestAddress, DWORD dwCountryCode)
```

Dials the specified dialable number on the specified call.

- Returns *dwRequestID* or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_OPERATIONFAILED
LINEERR_INVALADDRESS	LINEERR_RESOURCEUNAVAIL
LINEERR_INVALCOUNTRYCODE	LINEERR_DIALBILLING
LINEERR_INVALCALLSTATE	LINEERR_DIALQUIET
LINEERR_ADDRESSBLOCKED	LINEERR_DIALDIALTONE
LINEERR_NOMEM	LINEERR_DIALPROMPT
LINEERR_OPERATIONUNAVAIL	

dwRequestID

Specifies the identifier of the asynchronous request.

hdCall

Specifies the service provider's handle to the call to be dialed. The call state of *hdCall* can be any state except *idle* and *disconnected*.

lpszDestAddress

Specifies the destination to be dialed using the standard dialable number format.

dwCountryCode

Specifies the country code of the destination. This is used by the implementation to select the call progress protocols for the destination address. If a value of zero is specified, a default call-progress protocol defined by the service provider is used. Note that this parameter is not validated by TAPI.DLL when this function is called.

The service provider returns LINEERR_INVALCALLSTATE if the current state of the call does not allow dialing.

The service provider carries out no dialing if it returns LINEERR_INVALADDRESS.

If the service provider returns LINEERR_DIALBILLING, LINEERR_DIALQUIET, LINEERR_DIALDIALTONE, or LINEERR_DIALPROMPT, it should perform none of the actions otherwise performed by **TSPI_lineDial**. For example, no partial dialing, and no going off-hook. This is because the service provider should pre-scan the number for unsupported characters first.

TSPI_lineDial is used for dialing on an existing call appearance; for example, call handles returned from **TSPI_lineMakeCall** with NULL as the *lpszDestAddress* or ending in ';', call handles returned from **TSPI_lineSetupTransfer** or **TSPI_lineSetupConference**. Note that **TSPI_lineDial** may be invoked multiple times in the course of dialing in the case of multi-stage dialing, if the line's device capabilities permit it.

Multiple addresses may be provided in a single dial string separated by CRLF. Service providers that provide inverse multiplexing can establish individual physical calls with each of the addresses, and return a single call handle to the aggregate of all calls to the application. All addresses would use the same country code.

Dialing is considered complete after the address has been accepted by the service provider; not after the call is finally connected. Service providers that provide inverse multiplexing may allow multiple addresses to be provided at once. The service provider must send **LINE_CALLSTATE** messages to TAPI to inform it about the progress of the call.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original

thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

TSPI_lineDrop

[New - Windows 95]

```
LONG TSPI_lineDrop(DRV_REQUESTID dwRequestID, HDRVCALL hdCall,
    LPCSTR lpsUserUserInfo, DWORD dwSize)
```

Drops or disconnects the specified call. User-to-user information can optionally be transmitted as part of the call disconnect. This function can be called by the application at any time. When **TSPI_lineDrop** returns, the call should be *idle*.

- Returns *dwRequestID* or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_OPERATIONFAILED
LINEERR_INVALCALLSTATE	LINEERR_RESOURCEUNAVAIL
LINEERR_NOMEM	LINEERR_USERUSERINFOTOOBIG
LINEERR_OPERATIONUNAVAIL	

dwRequestID

Specifies the identifier of the asynchronous request.

hdCall

Specifies the service provider's handle to the call to be dropped. The call state of *hdCall* can be any state except *idle*.

lpsUserUserInfo

This pointer is only valid if **dwUserUserInfoSize** is non-zero. It specifies a far pointer to a string containing user-to-user information to be sent to the remote party as part of the call disconnect. This pointer is NULL if no user-to-user information is to be sent. User-to-user information is only sent if supported by the underlying network (see **LINEDEVCAPS**).

dwSize

Specifies the size in bytes of the user-to-user information in *lpsUserUserInfo*. If *lpsUserUserInfo* is Null dwSize should be ignored.

The service provider returns LINEERR_INVALCALLSTATE if the current state of the call does not allow the call to be dropped.

When invoking **TSPI_lineDrop**, related calls may sometimes be affected as well. For example, dropping a conference call may drop all individual participating calls. **LINE_CALLSTATE** messages are sent to TAPI.DLL for all calls whose call state is affected. A dropped call will typically transition to the *idle* state. Invoking **TSPI_lineDrop** on a call in the *offering* state rejects the call. Not all telephone networks provide this capability.

In situations where the call to be dropped is a consultation call established during transfer or conference call establishment, the original call that was placed in the *OnHoldPending* state is reconnected to and it will typically re-enter the *connected* call state.

TAPI.DLL has the option to send user-to-user information at the time of the drop. Even if user-to-user information can be sent, there is no guarantee that the network will deliver this information to the remote party.

Note that in various bridged or party line configurations when multiple parties are on the call, **TSPI_lineDrop** may not actually clear the call.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

TSPI_lineForward

[New - Windows 95]

```
LONG TSPI_lineForward(DRV_REQUESTID dwRequestID, HDRVLINE hdLine,
    DWORD bAllAddresses, DWORD dwAddressID,
    LPLINEFORWARDLIST const lpForwardList, DWORD dwNumRingsNoAnswer,
    HTAPICALL htConsultCall, LPHDRVCALL lphdConsultCall,
    LPLINECALLPARAMS const lpCallParams)
```

Forwards calls destined for the specified address on the specified line, according to the specified forwarding instructions. When an originating address (*dwAddressID*) is forwarded, the specified incoming calls for that address are deflected to the other number by the switch. This function provides a combination of forward and do-not-disturb features. This function can also cancel specific forwarding currently in effect.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALLINEHANDLE	LINEERR_NOMEM
LINEERR_INVALADDRESS	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALADDRESSID	LINEERR_OPERATIONFAILED
LINEERR_INVALCOUNTRYCODE	LINEERR_RESOURCEUNAVAIL
LINEERR_INVALPARAM	LINEERR_STRUCTURETOOSMALL

dwRequestID

Specifies the identifier of the asynchronous request.

hdLine

Specifies the service provider's handle to the line to be forwarded.

bAllAddresses

Specifies whether all originating addresses on the line or just the one specified is to be forwarded. If TRUE, all addresses on the line are forwarded and *dwAddressID* is ignored; if FALSE, only the address specified as *dwAddressID* is forwarded. Note that this parameter is not validated by TAPI.DLL when this function is called.

dwAddressID

Specifies the address on the specified line whose incoming calls are to be forwarded. This parameter is ignored if *bAllAddresses* is TRUE. Note that this parameter is not validated by TAPI.DLL when this function is called.

lpForwardList

Specifies a far pointer to a variably sized data structure of type **LINEFORWARDLIST** that describes the specific forwarding instructions.

dwNumRingsNoAnswer

Specifies the number of rings before an incoming call is considered a "no answer." If *dwNumRingsNoAnswer* is out of range, the actual value is set to the nearest value in the allowable range. Note that this parameter is not validated by TAPI.DLL when this function is called.

htConsultCall

Specifies TAPI.DLL's handle to a new call, if such a call must be created by the service provider. In some telephony environments, forwarding a call has the side effect of creating a consultation call used to consult the party that is being forwarded to. In such an environment, the service provider creates the new consutation call and must save this value and use it in all subsequent calls to the **LINEEVENT** procedure reporting events on the call. If no consultation call is created, this value can be ignored by the service provider.

lphdConsultCall

Specifies a far pointer to an HDRVCALL representing the service provider's identifier for the call. In telephony environments where forwarding a call has the side effect of creating a consultation call used to consult the party that is being forwarded to, the service provider must fill this location with its

handle for the call before this procedure returns. The service provider is permitted to do callbacks regarding the new call before it returns from this procedure. If no consultation call is created, the HDRVCALL must be left NULL.

lpCallParams

Specifies a far pointer to a structure of type **LINECALLPARAMS**. This pointer is ignored by the service provider unless **lineForward** requires the establishment of a call to the forwarding destination (and *lphdConsultCall* is returned, in which case *lpCallParams* is optional). If NULL, default call parameters are used. Otherwise, the specified call parameters are used for establishing *htConsultCall*.

The service provider returns LINEERR_INVALPARAM if the specified forward list parameter contains invalid information.

The service provider carries out no dialing if it returns LINEERR_INVALADDRESS.

The service provider returns “success” to this function to indicate only that the request has been accepted by the service provider, not that forwarding is set up at the switch. A **LINE_ADDRESSSTATE** (forwarding) message is sent to provide that forwarding has been set up at the switch.

Forwarding of the address(es) remains in effect until this function is called again. The most recent forwarding list replaces any old one in effect. If this function is called, specifying a NULL pointer as *lpForwardList*, the service provider should cancel any forwarding that is being performed at that time. If a NULL destination address is specified for an entry in the forwarding list, the operation acts as a “do-not-disturb.”

Forwarding status of an address may also be affected externally, for example, through administrative actions at the switch, or by a user from another station. It may not be possible for the service provider to be aware of this state change, and may not be able to keep in sync with the forwarding state known to the switch. The provider should always indicate what it knows to be true, and indicate that the forwarding state is “unknown” otherwise.

Because a service provider may not know the forwarding state of the address with no doubt (that is, it may have been forwarded or unforwarded in an unknown way), **TSPI_lineForward** will succeed unless it fails to set the new forwarding instructions. In other words, a request that all forwarding be canceled at a time that there is no forwarding in effect will be successful. This is because there is no “unforwarding”—you can only invoke a new set of forwarding instructions.

The success or failure of this operation does not depend on the previous set of forwarding instructions, and the same is true when setting different forwarding instructions. If necessary, the provider should “unforward everything” prior to setting the new forwarding instructions. Since this may take time in analog telephony environments, a provider may also want to compare the current forwarding with the new one, and only issue instructions to the switch to get to the final state (leaving unchanged forwarding unaffected).

Invoking **TSPI_lineForward** when **LINEFORWARDLIST** has **dwNumEntries** set to zero has the same effect as providing a NULL *lpForwardList* parameter; it cancels all forwarding currently in effect.

Since the NULL value returned into *lphdConsultCall* is the only way for TAPI.DLL to determine whether the service provider created a consultation call, the service provider can not use NULL as a call handle.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

This function differs from the corresponding TAPI function in that it follows the TSPI model for beginning the lifetime of a call. TAPI.DLL and the service provider exchange opaque handles representing the call with one another. In addition, the service provider is permitted to do callbacks for the new call before it returns from this procedure. In any case, the service provider must also treat the handle it returned as “not yet valid” until after the matching **ASYNC_COMPLETION** message reports success. In other words, it must not issue any messages for the new call or include it in call counts in messages or status data structures for the line.

TSPI_lineGatherDigits

[New - Windows 95]

```
LONG TSPI_lineGatherDigits(HDRVCALL hdCall, DWORD dwEndToEndID,
    DWORD dwDigitModes, LPSTR lpsDigits, DWORD dwNumDigits,
    LPCSTR lpszTerminationDigits, DWORD dwFirstDigitTimeout,
    DWORD dwInterDigitTimeout)
```

Initiates the buffered gathering of digits on the specified call. TAPI.DLL specifies a buffer in which to place the digits and the maximum number of digits to be collected.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_RESOURCEUNAVAIL
LINEERR_INVALCALLSTATE	LINEERR_NOMEM
LINEERR_INVALTIMEOUT	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALDIGITMODE	LINEERR_OPERATIONFAILED
LINEERR_INVALDIGITS	LINEERR_RESOURCEUNAVAIL
LINEERR_INVALPARAM	

Digit collection can be terminated in the following ways:

- The requested number of digits may have been collected.
- One of the digits detected matches a digit in *szTerminationDigits* before the specified number of digits has been collected. The detected termination digit is also placed in the buffer and the partial buffer is returned.
- One of the timeouts expires. The *dwFirstDigitTimeout* expires if the first digit is not received in this time period. The *dwInterDigitTimout* expires if the second, third, (and so forth) digit is not received within that time period from the previously detected digit, and a partial buffer is returned.
- Calling this operation again while collection is in progress. The old collection session is terminated and the contents of the old buffer is undefined. The mechanism for canceling digit gathering without initiating another operation is by invoking this operation with *lpsDigits* equal to NULL.

hdCall

Specifies the service provider's handle to the call on which digit gathering is to be performed. The call state of *hdCall* can be any state except *idle*.

dwEndToEndID

Specifies a unique, uninterpreted identifier of the request for its entire lifetime, that is, until the matching **LINE_GATHERDIGITS** message is sent. The service provider includes this identifier as one of the parameters in the message.

dwDigitModes

Specifies the digit mode(s) that are to be monitored. This parameter uses one or more of the following LINEDIGITMODE_ contants:

LINEDIGITMODE_PULSE

Detect digits as audible clicks that are the result of the use of rotary pulse sequences. Valid digits for pulse mode are '0' through '9'.

LINEDIGITMODE_DTMF

Detect digits as DTMF tones. Valid digits for DTMF mode are '0' through '9', 'A', 'B', 'C', 'D', '*', '#'.

lpsDigits

Specifies a far pointer to the buffer where detected digits are to be stored as ASCII characters. The service provider may, but is not required to, place digits in the buffer one at a time as they are collected. When the **LINE_GATHERDIGITS** message is sent, the content of the buffer must be complete. If *lpsDigits* is specified as NULL the digit gathering currently in progress on the call is canceled and the *dwNumDigits* parameter is ignored. Otherwise, *lpsDigits* is assumed to have room for *dwNumDigits* digits.

dwNumDigits

Specifies the number of digits to be collected before a **LINE_GATHERDIGITS** message is sent to

TAPI.DLL. *dwNumDigits* is ignored when *lpsDigits* is NULL. This function must return a LINEERR_INVALPARAM if *dwNumDigits* is zero.

lpszTerminationDigits

Specifies a NULL-terminated string of termination digits as ASCII characters. If one of the digits in the string is detected, that termination digit is appended to the buffer, digit collection is terminated and the **LINE_GATHERDIGITS** message is sent to TAPI.DLL.

Valid characters for pulse mode are '0' through '9'. Valid characters for DTMF mode are '0' through '9', 'A', 'B', 'C', 'D', '*', '#'. If this pointer is NULL, or if it points to an empty string, the function behaves as though no termination digits were supplied.

dwFirstDigitTimeout

Specifies the time duration in milliseconds in which the first digit is expected. If the first digit is not received in this timeframe, digit collection is terminated and a **LINE_GATHERDIGITS** message is sent to TAPI.DLL. A single NULL character is written to the buffer, indicating no digits were received and the first digit timeout terminated digit gathering. The call's line device capabilities specifies the valid range for this parameter or indicates that timeouts are not supported. Note that this parameter is not validated by TAPI.DLL when this function is called.

dwInterDigitTimeout

Specifies the maximum time duration in milliseconds between consecutive digits. If no digit is received in this timeframe, digit collection is terminated and a **LINE_GATHERDIGITS** message is sent to TAPI.DLL. A single NULL character is written to the buffer, indicating that an interdigit timeout terminated digit gathering. The **LINEDEVCAPS** structure must specify the the valid range for this parameter or indicate that timeouts are not supported. Note that this parameter is not validated by TAPI.DLL when this function is called.

The service provider returns LINEERR_INVALPARAM if the *dwNumDigits* parameter is invalid.

This function returns zero (success) if digit collection has been correctly initiated; not when digit collection has terminated. In all cases where a partial buffer is returned, valid digits (if any) are followed by an ASCII NULL character.

Although this function can be invoked in any call state except *idle*, digits can typically only be gathered while the call is in the *connected* state.

The message **LINE_GATHERDIGITS** is normally sent when the digit buffer becomes filled. It is also sent when partial buffers are returned because of timeouts or matching termination digits, or when the request is canceled via another **TSPI_lineGatherDigits** request on the call. Only one gather digits request can be active at a time. It is the responsibility of the service provider to terminate any outstanding gathering operation with a **LINE_GATHERDIGITS** message when **TSPI_lineGatherDigits** is called.

TAPI.DLL can use **TSPI_lineMonitorDigits** to enable or disable unbuffered digit detection. Each time a digit is detected in this fashion, a **LINE_MONITORDIGITS** message is sent to TAPI.DLL. Both buffered (gather digits) and unbuffered digit detection can be enabled for the same call simultaneously.

The service provider is allowed some variation in the quality of timing it uses for this function, including not doing timings at all. The quality of timing is reported in **LINEDEVCAPS**, in the fields **dwGatherDigitsMinTimeout** and **dwGatherDigitsMaxTimeout**.

The corresponding function at the TAPI level does not include the formal parameter *dwEndToEndID*. At that level, there is no end-to-end marking. TAPI.DLL uses end-to-end marking at the TSPI level to distinguish one **TSPI_lineGatherDigits** request from another.

TSPI_lineGenerateDigits

[New - Windows 95]

```
LONG TSPI_lineGenerateDigits(HDRVCALL hdCall, DWORD dwEndToEndID,
    DWORD dwDigitMode, LPCSTR lpszDigits, DWORD dwDuration)
```

Initiates the generation of the specified digits on the specified call as in-band tones using the specified signaling mode. Invoking this function while digit or tone generation is in progress aborts the current digit or tone generation. Passing a NULL value for *lpsDigits* generates no new digits. Note that only one inband generation request at a time (tone generation or digit generation) is allowed to be in progress per call.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_NOMEM
LINEERR_INVALCALLSTATE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALDIGITMODE	LINEERR_OPERATIONFAILED
LINEERR_RESOURCEUNAVAIL	LINEERR_RESOURCEUNAVAIL

hdCall

Specifies the handle to the call on which digit generation is to be done.

dwEndToEndID

This unique request ID should be stored by the service provider and passed back as *dwParam2* to the **LINEEVENT** procedure when the digit generation is completed.

dwDigitMode

Indicates the format to be used for signaling these digits. This paramter is allowed to have only one of the following LINEDIGITMODE_ constants set:

LINEDIGITMODE_PULSE

Uses pulse/rotary for digit signaling. Valid digits for pulse mode are ‘0’ through ‘9’ and ‘,’ (comma).

LINEDIGITMODE_DTMF

Uses DTMF tones for digit signaling. Valid digits for DTMF mode are ‘0’ through ‘9’, ‘A’, ‘B’, ‘C’, ‘D’, ‘*’, ‘#’, and ‘,’ (comma).

lpszDigits

Specifies a far pointer to a NULL terminated character buffer that contains the digits to be generated. A comma injects an extra delay between the signaling of the previous and next digits it separates. The duration of this pause is configuration-defined. The line’s device capabilities indicates what this duration is. Multiple commas may be used to inject longer pauses. Invalid digits are ignored during the generation, rather than being reported as an error.

dwDuration

Specifies both the duration in milliseconds of DTMF digits and pulse and DTMF inter-digit spacing. A value of zero will use a default value. *dwDuration* must be within the range specified by **MinDialParams** to **MaxDialParams** in **LINEDEVCAPS**. If out of range, the actual value is set by the service provider to the nearest value in the range. Note that this parameter is not validated by TAPI.DLL when this function is called.

The call state of *hdCall* can be any state.

The **TSPI_lineGenerateDigits** function is considered to have completed successfully when the digit generation has been successfully initiated; not when all digits have been generated.

After all digits in *lpsDigits* have been generated, or after digit generation has been aborted or canceled, a **LINE_GENERATE** message is sent to TAPI.DLL.

Note that only one inband generation request (tone generation or digit generation) is allowed to be in progress per call. This implies that if digit generation is currently in progress on a call, invoking either **TSPI_lineGenerateDigits** or **TSPI_lineGenerateTone** will cancel the digit generation. It is the responsibility of the service provider to terminate any digit generation in progress when a subsequent **TSPI_lineGenerateDigits** or **TSPI_lineGenerateTone** is invoked. Invoking **TSPI_lineGenerateDigits** with *lpszDigits* set to NULL cancels any current digit (or tone) generation.

Comparison

The corresponding function at the TAPI level does not include the formal parameter *dwEndToEndID*. At that level, there is no end-to-end marking. TAPI.DLL uses end-to-end marking at the TSPI level to disambiguate one **TSPI_lineGenerateDigits** request from another.

TSPI_lineGenerateTone

[New - Windows 95]

```
LONG TSPI_lineGenerateTone(HDRVCALL hdCall, DWORD dwEndToEndID,
    DWORD dwToneMode, DWORD dwDuration, DWORD dwNumTones,
    LPLINEGENERATETONE const lpTones)
```

Generates the specified tone inband over the specified call. Invoking this function with a zero for *dwToneMode* aborts any tone generation currently in progress on the specified call. Invoking **TSPI_lineGenerateTone** or **TSPI_lineGenerateDigits** while tone generation is in progress aborts the current tone generation or digit generation in progress and initiates the generation of the newly specified tone or digits.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

  LINEERR_INVALCALLHANDLE
  LINEERR_INVALCALLSTATE
  LINEERR_INVALTONEMODE
  LINEERR_INVALTONE
  LINEERR_RESOURCEUNAVAIL
  LINEERR_NOMEM
  LINEERR_OPERATIONUNAVAIL
  LINEERR_OPERATIONFAILED
  LINEERR_RESOURCEUNAVAIL

hdCall

Specifies the service provider's handle to the call on which tone generation is to be performed. The call state of *hdCall* can be any state except *idle*.

dwEndToEndID

Specifies a unique, uninterpreted identifier of the request for its entire lifetime, that is, until the matching **LINE_GENERATE** message is sent. The service provider includes this identifier as one of the parameters in the message.

dwToneMode

Defines the tone to be generated. Tones can be either standard or custom. A custom tone is composed of a set of arbitrary frequencies. A small number of standard tones are predefined. The duration of the tone is specified by **dwDuration** for both standard and custom tones. If *dwToneMode* is set to zero, any digit or tone generation then in progress is cancelled. This parameter uses the following LINETONEMODE_ constants. Only one flag value may be set at a time.

LINETONEMODE_CUSTOM
The tone is a custom tone, defined by the specified frequencies.

LINETONEMODE_RINGBACK
The tone to be generated is ring tone. The exact ringback tone is service provider defined.

LINETONEMODE_BUSY
The tone is a standard (station) busy tone. The exact busy tone is service provider defined.

LINETONEMODE_BEEP
The tone is a beep, as used to announce the beginning of a recording. The exact beep tone is service provider defined.

LINETONEMODE_BILLING
The tone is billing information tone such as a credit card prompt tone. The exact billing tone is service provider defined.

dwDuration

Specifies the duration in milliseconds during which the tone should be sustained. A value of zero for *dwDuration* uses a default duration for the specified tone. Default values are:

CUSTOM:	infinite
RINGBACK:	infinite
BUSY:	infinite
BEEP:	infinite
BILLING:	fixed (single cycle)

Note that this parameter is not validated by TAPI.DLL when this function is called.

dwNumTones

Specifies the number of entries in the *lpTones* array. This field is ignored if *dwToneMode* is not equal to LINETONEMODE_CUSTOM.

lpTones

Specifies a far pointer to a **LINEGENERATETONE** array that specifies the tone's components. This parameter is ignored for non-custom tones. If *lpTones* is a multi-frequency tone, the various tones are played simultaneously.

TSPI_lineGenerateTone returns zero (success) when the tone generation has been successfully initiated; not when the generation of the tone is done. The function allows the inband generation of several predefined tones, such as ring back, busy tones, and beep. It also allows for the fabrication of custom tones by specifying their component frequencies, cadence and volume if this is supported by the service provider. Since these tones are generated as inband tones, the call would typically have to be in the *connected* state for tone generation to be effective. When the generation of the tone is complete, or when tone generation is canceled, a **LINE_GENERATE** message is sent to TAPI.DLL.

Note that only one inband generation request (tone generation or digit generation) is allowed to be in progress per call. This implies that if tone generation is currently in progress on a call, invoking either **TSPI_lineGenerateDigits** or **TSPI_lineGenerateTone** will cancel the tone generation. It is the responsibility of the service provider to terminate any tone generation in progress when a subsequent **TSPI_lineGenerateDigits** or **TSPI_lineGenerateTone** is invoked.

Comparison

The corresponding function at the TAPI level does not include the formal parameter *dwEndToEndID*. At that level, there is no end-to-end marking. TAPI.DLL uses end-to-end marking at the TSPI level to distinguish one **TSPI_lineGenerateTone** request from another.

TSPI_lineGetAddressCaps

[New - Windows 95]

```
LONG TSPI_lineGetAddressCaps(DWORD dwDeviceID, DWORD dwAddressID,
    DWORD dwTSPIVersion, DWORD dwExtVersion,
    LPLINEADDRESSCAPS lpAddressCaps)
```

Queries the specified address on the specified line device to determine its telephony capabilities.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INCOMPATIBLEAPIVERSION	LINEERR_NOMEM
LINEERR_INCOMPATIBLEEXTVERSION	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALADDRESSID	LINEERR_OPERATIONFAILED
LINEERR_NODRIVER	LINEERR_RESOURCEUNAVAIL

dwDeviceID

Specifies the line device containing the address to be queried.

dwAddressID

Specifies the address on the given line device whose capabilities are to be queried. Note that this parameter is not validated by TAPI.DLL when this function is called.

dwTSPIVersion

Specifies the version number of the Telephony SPI to be used. The high-order word contains the major version number; the low-order word contains the minor version number.

dwExtVersion

Specifies the version number of the service provider-specific extensions to be used. This number will be zero if no device-specific extensions are to be used. Otherwise, the high-order word contains the major version number; the low-order word contain the minor version number. Note that this parameter is not validated by TAPI.DLL when this function is called.

lpAddressCaps

Specifies a far pointer to a variably sized structure of type **LINEADDRESSCAPS**. Upon successful completion of the request, this structure is filled with address capabilities information.

The line device IDs supported by a particular driver are numbered sequentially starting the value of *dwLineDeviceIDBase* that is passed into the **TSPI_providerInit** function.

The service provider should fill in all the fields of the **LINEADDRESSCAPS** data structure, except for **dwTotalSize**, which is filled in by TAPI.DLL. The service provider must *not* overwrite the **dwTotalSize** field.

TSPI_lineGetAddressID

[New - Windows 95]

```
LONG TSPI_lineGetAddressID(HDRVLINE hdLine, LPDWORD lpdwAddressID,
    DWORD dwAddressMode, LPCSTR lpsAddress, DWORD dwSize)
```

Returns the address ID associated with address in a different format on the specified line.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALLINEHANDLE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALADDRESS	LINEERR_OPERATIONFAILED
LINEERR_NOMEM	LINEERR_RESOURCEUNAVAIL

hdLine

Specifies the service provider's handle to the line whose address is to be retrieved.

lpdwAddressID

Specifies a far pointer to a DWORD-sized memory location where the address ID is returned.

dwAddressMode

Specifies the address mode of the address contained in *lpsAddress*. *dwAddressMode* is allowed to have only the following LINEADDRESSMODE_ flag set:

LINEADDRESSMODE_DIALABLEADDR

The address is specified using its dialable address. *lpsAddress* is the dialable address format.

lpsAddress

Specifies a far pointer to a data structure holding the address assigned to the specified line device. The format of the address is determined by *dwAddressMode* parameter. If it is LINEADDRESSMODE_DIALABLEADDR, the *lpsAddress* uses the common dialable number format and is NULL-terminated.

dwSize

Specifies the size of the address contained in *lpsAddress*. The parameter *dwSize* must be set to the length of the string (plus one for the NULL) if LINEADDRESSMODE_DIALABLEADDR is used. If an extended LINEADDRESSMODE is used, the length should match the size of whatever is actually passed in (the DLL checks to be sure it can read the number of bytes specified from the pointer given).

This function is used to map a phone number (address) assigned to a line device back to its *dwAddressID* (in the range 0 to the number of addresses minus one) that is returned in the line's device capabilities.

TSPI_lineGetAddressStatus

[New - Windows 95]

```
LONG TSPI_lineGetAddressStatus(HDRVLINE hdLine, DWORD dwAddressID,
    LPLINEADDRESSSTATUS lpAddressStatus)
```

Queries the specified address for its current status.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALLINEHANDLE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALADDRESSID	LINEERR_OPERATIONFAILED
LINEERR_NOMEM	LINEERR_RESOURCEUNAVAIL

hdLine

Specifies the service provider's handle to the line containing the address to be queried.

dwAddressID

Specifies an address on the given open line device. This is the address to be queried. Note that this parameter is not validated by TAPI.DLL when this function is called.

lpAddressStatus

Specifies a far pointer to a variably sized data structure of type **LINEADDRESSSTATUS**.

The service provider should fill in all the fields of the **LINEADDRESSSTATUS** data structure, except for **dwTotalSize**, which is filled in by TAPI.DLL. The service provider must *not* overwrite the **dwTotalSize** field.

TSPI_lineGetCallAddressID

[New - Windows 95]

```
LONG TSPI_lineGetCallAddressID(HDRVCALL hdCall, LPDWORD lpdwAddressID)
```

Retrieves the address ID for the indicated call.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_NOMEM	LINEERR_OPERATIONFAILED
LINEERR_OPERATIONUNAVAIL	LINEERR_RESOURCEUNAVAIL

hdCall

Specifies the service provider's handle to the call whose address ID is to be retrieved. The call state of *hdCall* can be any.

lpdwAddressID

Specifies a far pointer to a DWORD into which the service provider writes the call's address ID.

If the service provider models lines as "pools" of channel resources and does inverse multiplexing of a call over several address IDs it should consistently choose one of these address IDs as the primary identifier reported by this function and in the **LINE_CALLINFO** data structure.

This function has no direct correspondence at the TAPI level. It gives TAPI.DLL sufficient information to implement the **lineGetNewCalls** function invoked with the LINECALLSELECT_ADDRESS option.

TSPI_lineGetCallInfo

[New - Windows 95]

```
LONG TSPI_lineGetCallInfo(HDRVCALL hdCall, LPLINECALLINFO lpCallInfo)
```

Returns detailed information about the specified call.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

  LINEERR_INVALCALLHANDLE LINEERR_OPERATIONFAILED
  LINEERR_NOMEM LINEERR_RESOURCEUNAVAIL
  LINEERR_OPERATIONUNAVAIL

hdCall

Specifies the service provider's handle to the call whose call information is to be retrieved. The call state of *hdCall* can be any state.

lpCallInfo

Specifies a far pointer to a variably sized data structure of type **LINECALLINFO**. Upon successful completion of the request, this structure is filled with call-related information.

The following table indicates which fields of the **LINECALLINFO** data structure are filled in by the service provider and which fields are filled in by TAPI.DLL. The service provider must preserve (it must *not* overwrite) the values filled in by TAPI.DLL.

Field Name	TAPI DLL	Service Provider
dwTotalSize;	X	
dwNeededSize;		X
dwUsedSize;		X
hLine;	X	
dwLineDeviceID;		X
dwAddressID;		X
dwBearerMode;		X
dwRate;		X
dwMediaMode;		X
dwAppSpecific;		X
dwCallID;		X
dwRelatedCallID;		X
dwCallParamFlags;		X
dwCallStates;		X
dwMonitorDigitModes;	X	
dwMonitorMediaModes;	X	
DialParams;		X
dwOrigin;		X
dwReason;		X
dwCompletionID;		X
dwNumOwners;	X	

dwNumMonitors;	X	
dwCountryCode;		X
dwTrunk;		X
dwCallerIDFlags;		X
dwCallerIDSize;		X
dwCallerIDOffset;		X
dwCallerIDNameSize;		X
dwCallerIDNameOffset;		X
dwCalledIDFlags;		X
dwCalledIDSize;		X
dwCalledIDOffset;		X
dwCalledIDNameSize;		X
dwCalledIDNameOffset;		X
dwConnectedIDFlags;		X
dwConnectedIDSize;		X
dwConnectedIDOffset;		X
dwConnectedIDNameSize;		X
dwConnectedIDNameOffset;		X
dwRedirectionIDFlags;		X
dwRedirectionIDSize;		X
dwRedirectionIDOffset;		X
dwRedirectionIDNameSize;		X
dwRedirectionIDNameOffset;		X
dwRedirectingIDFlags;		X
dwRedirectingIDSize;		X
dwRedirectingIDOffset;		X
dwRedirectingIDNameSize;		X
dwRedirectingIDNameOffset;		X
dwAppNameSize;	X	
dwAppNameOffset;	X	
dwDisplayableAddressSize;	X	
dwDisplayableAddressOffset;	X	
dwCalledPartySize;	X	
dwCalledPartyOffset;	X	
dwCommentSize;	X	

dwCommentOffset;	X	
dwDisplaySize;		X
dwDisplayOffset;		X
dwUserUserInfoSize;		X
dwUserUserInfoOffset;		X
dwHighLevelCompSize;		X
dwHighLevelCompOffset;		X
dwLowLevelCompSize;		X
dwLowLevelCompOffset;		X
dwChargingInfoSize;		X
dwChargingInfoOffset;		X
dwTerminalModesSize;		X
dwTerminalModesOffset;		X
dwDevSpecificSize;		X
dwDevSpecificOffset;		X

TAPI.DLL fills in the size and offset fields for the **dwAppNameSize/Offset, dwCalledPartySize/Offset,** and **dwCommentSize/Offset** fields and updates the value in **dwUsedSize** to reflect these after calling the service provider.

If the service provider models lines as “pools” of channel resources and does inverse multiplexing of a call over several address IDs it should consistently choose one of these address IDs as the primary identifier reported by this function in the **LINECALLINFO** data structure.

TSPI_lineGetCallStatus

[New - Windows 95]

```
LONG TSPI_lineGetCallStatus(HDRVCALL hdCall,
    LPLINECALLSTATUS lpCallStatus)
```

Returns the current status of the specified call.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

  LINEERR_INVALCALLHANDLE LINEERR_OPERATIONFAILED
  LINEERR_NOMEM LINEERR_RESOURCEUNAVAIL
  LINEERR_OPERATIONUNAVAIL

hdCall

Specifies the service provider's handle to the call to be queried for its status. The call state of *hdCall* can be any state.

lpCallStatus

Specifies a far pointer to a variably sized data structure of type **LINECALLSTATUS**. This structure is filled with call status information.

The following table indicates which fields of the **LINECALLSTATUS** data structure are filled in by the service provider and which fields are filled in by TAPI.DLL. The service provider must preserve (it must *not* overwrite) the values filled in by TAPI.DLL.

Field Name	TAPI DLL	Service Provider
dwTotalSize;	X	
dwNeededSize;		X
dwUsedSize;		X
dwCallState;		X
dwCallStateMode;		X
dwCallPrivilege;	X	
dwCallFeatures;		X
dwDevSpecificSize;		X
dwDevSpecificOffset;		X

TSPI_lineGetCallStatus returns the dynamic status of a call, whereas **TSPI_lineGetCallInfo** returns primarily static information about a call. Call status information includes the current call state, detailed mode information related to the call while in this state (if any), as well as a list of the available TSPI functions TAPI.DLL can invoke on the call while the call is in this state.

TSPI_lineGetDevCaps

[New - Windows 95]

```
LONG TSPI_lineGetDevCaps(DWORD dwDeviceID, DWORD dwTSPIVersion,
    DWORD dwExtVersion, LPLINEDEVCAPS lpLineDevCaps)
```

Queries a specified line device to determine its telephony capabilities. The returned information is valid for all addresses on the line device.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INCOMPATIBLEAPIVERSION	LINEERR_OPERATIONUNAVAIL
LINEERR_INCOMPATIBLEEXTVERSION	LINEERR_OPERATIONFAILED
LINEERR_NODRIVER	LINEERR_RESOURCEUNAVAIL
LINEERR_NOMEM	

dwDeviceID

Specifies the line device to be queried.

dwTSPIVersion

Specifies the negotiated TSPI version number. This value has already been negotiated for this device through the **TSPI_lineNegotiateTSPIVersion** function.

dwExtVersion

Specifies the negotiated extension version number. This value has already been negotiated for this device through the **TSPI_lineNegotiateExtVersion** function. Note that this parameter is not validated by TAPI.DLL when this function is called.

lpLineDevCaps

Specifies a far pointer to a variably sized structure of type **LINEDEVCAPS**. Upon successful completion of the request, this structure is filled with line device capabilities information.

Line device ID numbering for a service provider is sequential from the value set by the *dwLineDeviceIDBase* parameter that is passed to the function **TSPI_providerInit**.

The *dwExtVersion* formal parameter indicates the version number of the extension information requested. If it is zero, no extension information is requested. If it is non-zero, it holds a value that has already been negotiated for this device with the function **TSPI_lineNegotiateExtVersion**. The service provider should fill in device- and vendor-specific extended information according to the extension version specified.

The service provider should fill in all the fields of the **LINEDEVCAPS** data structure, except for **dwTotalSize**, which is filled in by TAPI.DLL. The service provider must *not* overwrite the **dwTotalSize** field.

The service provider must fill in all fields of the **LINETERMCAPS** data structure(s) embedded in the varying part of the **LINEDEVCAPS** data structure.

TSPI_lineGetDevConfig

[New - Windows 95]

```
LONG TSPI_lineGetDevConfig (DWORD dwDeviceID,
    LPVARSTRING lpDeviceConfig, LPCSTR lpszDeviceClass)
```

Returns a data structure object, the contents of which are specific to the line (service provider) and device class, giving the current configuration of a device associated one-to-one with the line device.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are:

LINEERR_INVALDEVICECLASS	LINEERR_NOMEM
LINEERR_INVALPOINTER	LINEERR_OPERATIONUNAVAIL
LINEERR_STRUCTURETOOSMALL	LINEERR_OPERATIONFAILED
LINEERR_NODRIVER	LINEERR_RESOURCEUNAVAIL

dwDeviceID

Specifies the line device to be configured.

lpDeviceConfig

Specifies a far pointer to a data structure of type **VARSTRING** where the device configuration structure of the associated device is returned. Upon successful completion of the request, the service provider fills this data structure with the device configuration. The **dwStringFormat** field in the **VARSTRING** structure must be set to STRINGFORMAT_BINARY.

lpszDeviceClass

Specifies a far pointer to a NULL-terminated ASCII string that specifies the device class of the device whose configuration is requested. Valid device class strings are the same as those specified for the **TSPI_lineGetID** function when it is applied to a “line” device (*dwSelect* has the value LINECALLSELECT_LINE).

The call state is device specific.

This function can be used to retrieve a data structure from the service provider which specifies the configuration of a device associated one-to-one with the line device. The *lpszDeviceClass* parameter selects which of among possibly several different classes of devices is to have its configuration retrieved. The set of supported classes is restricted to those whose devices correspond one-to-one with the line device.

A service provider should typically allow the “tapi/line” device class under this function. It would retrieve parameters that have “line” scope, such as the list of addresses in this line, the list of physical hardware devices such as COMM ports corresponding to the addresses, maximum number of concurrent calls (if configurable), and so on.

In general, this function would not allow media-related device classes such as mci waveaudio, low level wave, or datamodem device classes, since these usually apply to a particular call or a particular address. Since there may be more than one of these per line device, the identification of the particular call or address simply by the line device ID parameter in this function would be ambiguous. An exception can be made for call-specific or address-specific device classes in cases where there is class configuration information that applies to the entire line device scope, such as initial defaults, and so on.

There are several reasons why “exceptional” support for call-specific and address-specific device classes is of only limited value under this function. First, since these classes may be ambiguous on multiple-address/multiple-call service providers, only a subset of service providers will support them. Applications are not likely to add a device-specific dependency on the inclusion of these classes in this function. Second, as higher-level media “classes” emerge that implement high-level protocols such as dial-in file system access in terms of low-level transport APIs, configuration for these classes will tend toward “instance” scope instead of “class” scope. The high-level media API will have to supply its own functions to configure call-specific or address-specific instances.

Whatever sort of devices and device classes this function supports, it can potentially affect two kinds of configuration information: permanent and temporary. Permanent information survives across different

“opens” of the line, and even across different “inits” of the service provider itself. Temporary information survives only within a unique “open” of the line. When the line is closed, any such temporary information that has been retrieved or set through **TSPI_lineSetDevConfig** may revert to default or undefined values. The caller can reliably retrieve any temporary configuration only by a sequence such as **TSPI_lineOpen**, **TSPI_lineConfigDialog**, **TSPI_lineGetDevConfig**. The caller can reliably set temporary configuration information retrieved by such a sequence through a sequence such as **TSPI_lineOpen**, **TSPI_lineSetDevConfig**. The temporary part of configuration remains stable only until the next **TSPI_lineConfigDialog**, **TSPI_lineSetDevConfig**, or **TSPI_lineClose**. The service provider must take care of storing any permanent part of the configuration, typically in an “INI” file, and reloading it whenever the service provider is initialized.

The exact format of the data contained within the structure returned by this function is specific to the line and device class API, is undocumented, and is undefined. The structure returned by this function cannot be directly accessed or manipulated by the application, but can only be stored intact and later used in **TSPI_lineSetDevConfig** to restore the settings. The structure also cannot necessarily be passed to other devices, even of the same device class (although this may work in some instances, it is not guaranteed). A service provider should put items in the data structure to allow it to be checked for consistency to guard against failures due to a client application passing incompatible information.

TSPI_lineGetExtensionID

[New - Windows 95]

```
LONG TSPI_lineGetExtensionID(DWORD dwDeviceID, DWORD dwTSPIVersion,
    LPLINEEXTENSIONID lpExtensionID)
```

Returns the extension ID that the service provider supports for the indicated line device.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

  LINEERR_NOMEM LINEERR_OPERATIONFAILED
  LINEERR_OPERATIONUNAVAIL LINEERR_RESOURCEUNAVAIL

dwDeviceID

Specifies the line device to be queried.

dwTSPIVersion

Specifies an interface version number that has already been negotiated for this device using **TSPI_lineNegotiateTSPIVersion**. This function operates according to the interface specification at this version level.

lpExtensionID

Specifies a far pointer to a structure of type **LINEEXTENSIONID**. If the service provider supports provider-specific extensions it fills this structure with the extension ID of these extensions. If the service provider does not support extensions, it fills this structure with all zeros. (Therefore, a valid extension ID cannot consist of all zeros.)

This function is typically called by TAPI.DLL in response to an application calling the **lineNegotiateAPIVersion** function. The result returned by the service provider should be appropriate for use in a subsequent call to **TSPI_lineNegotiateExtVersion**. An extension ID of all zeros is not a legal extension ID, since the all-zeros value is used to indicate that the service provider does not support extensions.

TSPI_lineGetIcon

[New - Windows 95]

```
LONG TSPI_lineGetIcon(DWORD dwDeviceID, LPCSTR lpszDeviceClass,
    LPHICON lphIcon)
```

Retrieves a service line device-specific icon for display to the user.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

  LINEERR_INVALDEVICECLASS LINEERR_OPERATIONFAILED
  LINEERR_NOMEM LINEERR_RESOURCEUNAVAIL
  LINEERR_OPERATIONUNAVAIL

dwDeviceID

Specifies the line device whose icon is requested.

lpszDeviceClass

Specifies a far pointer to a NULL-terminated string that identifies a device class name. This device class allows the caller to select an icon specific to that device class. This parameter is optional and can be left NULL, in which case the highest level icon associated with the line device rather than a specified media stream device would be selected.

Permitted strings are the same as for **TSPI_lineGetID**. For example, if the line supports the Comm API, passing “COMM” as *lpszDeviceClass* causes the provider to return an icon related specifically to the Comm device functions of the service provider.

lphIcon

Specifies a far pointer to a memory location in which the handle to the icon is returned.

The provider should return a handle (in the DWORD pointed to by *lphIcon*) to an icon resource (obtained from the Windows **LoadIcon** function) associated with the specified line.

A provider may choose to support many icons (selected by *lpszDeviceClass* and/or line number), a single icon (such as for the manufacturer, which would be returned for all **TSPI_lineGetIcon** requests regardless of the *lpszDeviceClass* selected), or no icons, in which case it sets the DWORD pointed to by *lphIcon* to NULL. TAPI.DLL examines the handle returned by the provider, and if the provider returns NULL, TAPI.DLL substitutes a generic Windows Telephony icon (the generic “line” icon).

TSPI_lineGetID

[New - Windows 95]

```
LONG TSPI_lineGetID(HDRVLINE hdLine, DWORD dwAddressID, HDRVCALL hdCall,
    DWORD dwSelect, LPVARSTRING lpDeviceID, LPCSTR lpszDeviceClass)
```

Returns a device ID for the specified device class associated with the selected line, address or call.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALLINEHANDLE	LINEERR_NOMEM
LINEERR_INVALADDRESSID	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALCALLHANDLE	LINEERR_OPERATIONFAILED
LINEERR_NODEVICE	LINEERR_RESOURCEUNAVAIL

hdLine

Specifies the service provider's handle to the line to be queried.

dwAddressID

Specifies an address on the given open line device. Note that this parameter is not validated by TAPI.DLL when this function is called.

hdCall

Specifies the service provider's handle to the call to be queried.

dwSelect

Specifies the whether the device ID requested is associated with the line, address or a single call. The *dwSelect* parameter can only have one of the following LINECALLSELECT_ flags set:

LINECALLSELECT_LINE
: Selects the specified line device. The *hdLine* parameter must be a valid line handle; *hdCall* and *dwAddressID* are unused.

LINECALLSELECT_ADDRESS
: Selects the specified address on the line. Both *hdLine* and *dwAddressID* must be valid; *hdCall* is unused.

LINECALLSELECT_CALL
: Selects the specified call. *hdCall* must be valid; *hdLine* and *dwAddressID* are both unused.

lpDeviceID

Specifies a far pointer to the memory location of type **VARSTRING** where the device ID is returned. Upon successful completion of the request, this location is filled with the device ID. The format of the returned information depends on the method used by the device class (API) for naming devices.

lpszDeviceClass

Specifies a far pointer to a NULL-terminated ASCII string that specifies the device class of the device whose ID is requested. Valid device class strings are those used in the SYSTEM.INI section to identify device classes (such as COM, Wave, and MCI.)

The service provide returns LINEERR_INVALLINEHANDLE if *dwSelect* is LINECALLSELECT_LINE or LINECALLSELECT_ADDRESS, and *hdLine* is invalid.

The service provider returns LINEERR_INVALCALLHANDLE if *dwSelect* is LINECALLSELECT_CALL and *hdCall* is invalid.

The service provider should support the "tapi/line" device class to allow applications to determine the real line device ID of an opened line. In that case, the variable data returned is the *dwDeviceID*.

Some common device class names that are currently used are listed below. The first portion of a name identifies the API used to manage the device class, and the second portion is typically used to identify a specific device type extension or subset of the overall API. Names are not case-sensitive:

Device	**Description**
comm	(generic serial-device API; comm port)
comm/datamodem	(reserved for use in a future version of Microsoft

	Windows)
wave	(low-level waveaudio)
mci/midi	(high-level midi sequencer)
mci/wave	(high-level wave device control)
tapi/line	(TAPI line device)
tapi/phone	(TAPI phone device)
ndis	(network driver interface)

A vendor that defines a device-specific media mode also needs to define the corresponding device-specific (proprietary) API to manage devices of the media mode. To avoid collisions on device class names assigned independently by different vendors, a vendor should select a name that uniquely identifies the vendor and then the media type; for example: “intel/video”.

The service provider should fill in all the fields of the **VARSTRING** data structure, except for **dwTotalSize**, which is filled in by TAPI.DLL. The service provider must *not* overwrite the **dwTotalSize** field.

TSPI_lineGetLineDevStatus

[New - Windows 95]

```
LONG TSPI_lineGetLineDevStatus(HDRVLINE hdLine,
    LPLINEDEVSTATUS lpLineDevStatus)
```

Queries the specified open line device for its current status. The information returned is global to all addresses on the line.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

  LINEERR_INVALLINEHANDLE LINEERR_OPERATIONFAILED
  LINEERR_NOMEM LINEERR_RESOURCEUNAVAIL
  LINEERR_OPERATIONUNAVAIL

hdLine

Specifies the service provider's handle to the line to be queried.

lpLineDevStatus

Specifies a far pointer to a variably sized data structure of type **LINEDEVSTATUS**. This structure is filled with the line's device status.

The following table indicates which fields of the **LINEDEVSTATUS** data structure are filled in by the service provider and which fields are filled in by TAPI.DLL. The service provider must preserve (it must *not* overwrite) the values filled in by TAPI.DLL.

Field Name	TAPI DLL	Service Provider
dwTotalSize;	X	
dwNeededSize;		X
dwUsedSize;		X
dwNumOpens;	X	
dwOpenMediaModes;	X	
dwNumActiveCalls;		X
dwNumOnHoldCalls;		X
dwNumOnHoldPendCalls;		X
dwLineFeatures;		X
dwNumCallCompletions;		X
dwRingMode;		X
dwSignalLevel;		X
dwBatteryLevel;		X
dwRoamMode;		X
dwDevStatusFlags;		X
dwTerminalModesSize;		X
dwTerminalModesOffset;		X
dwDevSpecificSize;		X
dwDevSpecificOffset;		X

TSPI_lineGetNumAddressIDs

[New - Windows 95]

```
LONG TSPI_lineGetNumAddressIDs(HDRVLINE hdLine,
    LPDWORD lpdwNumAddressIDs)
```

Retrieves the number of address IDs supported on the indicated line.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_NOMEM	LINEERR_OPERATIONFAILED
LINEERR_OPERATIONUNAVAIL	LINEERR_RESOURCEUNAVAIL

hdLine

Specifies the handle to the line for which the number of address IDs is to be retrieved.

lpdwNumAddressIDs

Specifies a far pointer to a DWORD. The location is filled with the number of address IDs supported on the indicated line. The value should be one or larger.

This function is called by TAPI.DLL in response to an application calling **lineSetNumRings**, **lineGetNumRings**, or **lineGetNewCalls**. TAPI.DLL uses the retrieved value to determine if the specified address ID is within the range supported by the service provider.

TSPI_lineHold

[New - Windows 95]

```
LONG TSPI_lineHold(DRV_REQUESTID dwRequestID, HDRVCALL hdCall)
```

Places the specified call on hold.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALCALLSTATE	LINEERR_OPERATIONFAILED
LINEERR_NOMEM	LINEERR_RESOURCEUNAVAIL

dwRequestID

Specifies the identifier of the asynchronous request.

hdCall

Specifies the service provider's handle to the call to be placed on hold. The call state of *hdCall* can be *connected*.

The call on hold is temporarily disconnected, allowing TAPI.DLL to use the line device for making or answering other calls. **TSPI_lineHold** performs a "hard hold" of the specified call, as opposed to a "consultation call." A call on hard hold typically cannot be transferred or included in a conference call, whereas a consultation call can. Consultation calls are initiated using **TSPI_lineSetupTransfer**, **TSPI_lineSetupConference**, or **TSPI_linePrepareAddToConference**.

After a call has been successfully placed on hold, the call state will typically transition to *onHold*. A held call is retrieved through **TSPI_lineUnhold**. While a call is on hold, the service provider may send **LINE_CALLSTATE** messages about state changes of the held call. For example, if the held party hangs up, the call state may transition to *disconnected*, and the service provider should send a **LINE_CALLSTATE** message indicating the new state.

TSPI_lineMakeCall

[New - Windows 95]

```
LONG TSPI_lineMakeCall(DRV_REQUESTID dwRequestID, HDRVLINE hdLine,
    HTAPICALL htCall, LPHDRVCALL lphdCall, LPCSTR lpszDestAddress,
    DWORD dwCountryCode, LPLINECALLPARAMS const lpCallParams)
```

Places a call on the specified line to the specified destination address. Optionally, call parameters can be specified if anything but default call setup parameters are requested.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_ADDRESSBLOCKED	LINEERR_INVALLINESTATE
LINEERR_BEARERMODEUNAVAIL	LINEERR_INVALRATE
LINEERR_CALLUNAVAIL	LINEERR_INVALLINEHANDLE
LINEERR_DIALBILLING	LINEERR_INVALADDRESS
LINEERR_DIALQUIET	LINEERR_INVALADDRESSID
LINEERR_DIALDIALTONE	LINEERR_INVALCALLPARAMS
LINEERR_DIALPROMPT	LINEERR_NOMEM
LINEERR_INUSE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALADDRESSMODE	LINEERR_OPERATIONFAILED
LINEERR_INVALBEARERMODE	LINEERR_RESOURCEUNAVAIL
LINEERR_INVALCOUNTRYCODE	LINEERR_RATEUNAVAIL
LINEERR_INVALMEDIAMODE	LINEERR_USERUSERINFOTOOBIG

dwRequestID

Specifies the identifier of the asynchronous request.

hdLine

Specifies the handle to the line on which the new call is to be originated.

htCall

Specifies TAPI.DLL's handle to the new call. The service provider must save this and use it in all subsequent calls to the **LINEEVENT** procedure reporting events on the call.

lphdCall

Specifies a far pointer to a call handle. The service provider must fill this location with its handle for the call before this procedure returns. This handle is ignored by TAPI.DLL if the function results in an error.

lpszDestAddress

Specifies a far pointer to the destination address. This follows the standard dialable number format. This pointer may be specified as NULL for non-dialed addresses (as with a hot phone, which always automatically connects to a predefined number) or when all dialing will be performed using **TSPI_lineDial**. In the latter case, **TSPI_lineMakeCall** will allocate an available call appearance which would typically remain in the *dialtone* state until dialing begins. Service providers that have inverse multiplexing capabilities may allow an application to specify multiple addresses at once.

dwCountryCode

Specifies the country code of the called party. If a value of zero is specified, a default will be used by the implementation.

lpCallParams

Specifies a far pointer to a **LINECALLPARAMS** structure. This structure allows TAPI.DLL to specify how it wants the call to be set up. If NULL is specified, a default 3.1kHz voice call is established, and an arbitrary origination address on the line is selected. This structure selects elements such as the call's bearer mode, data rate, expected media mode, origination address, blocking of caller ID information, and dialing parameters.

The service provider returns LINEERR_INVALLINESTATE if the line is currently not in a state in which this operation can be performed. A list of currently valid operations can be found in the **dwLineFeatures** field (of the type **LINEFEATURE**) in the **LINEDEVSTATUS** structure. (Calling **TSPI_lineGetLineDevStatus** updates the information in **LINEDEVSTATUS**.)

If the service provider returns LINEERR_DIALBILLING, LINEERR_DIALQUIET, LINEERR_DIALDIALTONE, or LINEERR_DIALPROMPT, it should perform none of the actions otherwise performed by **TSPI_lineMakeCall**. For example, no partial dialing, and no going off-hook. This is because the service provider should pre-scan the number for unsupported characters first.

After **TSPI_lineMakeCall** returns a SUCCESS reply callback message to the application, the service provider must send **LINE_CALLSTATE** messages to the *lpfnEventProc* passed in **TSPI_lineOpen** to notify TAPI.DLL about the progress of the call. A typical reported sequence may be dialtone, dialing, proceeding, ringback, and *connected*; The first state reported will not necessarily be LINECALLSTATE_DIALTONE. The service provider chooses how many of these states are reported. It is recommended that as many as possible are sent, so that applications can take appropriate actions.

The service provider initially does media monitoring on the new call for at least the set of media modes that were monitored for on the line.

If the dial string is NULL, the service provider should take the line to the dialtone state (for a Comm-based service provider, this would involve ATD) and send a call-state message indicating LINECALLSTATE_DIALTONE.

If the dial string ends in ‘;’ (a semicolon), the service provider should dial the partial number nad send the LINECALLSTATE_DIALING message. This call will be completed by calls to TSPI_lineDial.

If the dial string contains characters (W, @, $, ?) that are not supported by the service provider, the service provider must scan the dial string and return (synchronously) an error corresponding to the first invalid character.

If the LINECALLPARAMFLAGS_IDLE flag is set, the service provider must check the current line status (the equivalent of going off-hook and sensing dialtone). If this IDLE flag is set and there is no dialtone, the function fails with the error LINEERR_CALLUNAVAIL. If the IDLE flag is not set, or there is a dialtone, the dialing can continue.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

This function differs from the corresponding TAPI function in that it follows the TSPI model for beginning the lifetime of a call. TAPI.DLL and the service provider exchange opaque handles representing the call with one another. In addition, the service provider is permitted to do callbacks for the new call before it returns from this procedure. In any case, the service provider must also treat the handle it returned as “not yet valid” until after the matching **ASYNC_COMPLETION** message reports success. In other words, it must not issue any **LINEEVENT** messages for the new call or include it in call counts in messages or status data structures for the line.

TSPI_lineMonitorDigits

[New - Windows 95]

```
LONG TSPI_lineMonitorDigits(HDRVCALL hdCall, DWORD dwDigitModes)
```

Enables and disables the unbuffered detection of digits received on the call. Each time a digit of the specified digit mode(s) is detected, a **LINE_MONITORDIGITS** message is sent to the application by TAPI.DLL, indicating which digit has been detected.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

  LINEERR_INVALCALLHANDLE
  LINEERR_INVALCALLSTATE
  LINEERR_INVALDIGITMODE
  LINEERR_NOMEM
  LINEERR_OPERATIONUNAVAIL
  LINEERR_OPERATIONFAILED
  LINEERR_RESOURCEUNAVAIL

hdCall

Specifies the handle to the call on which digits are to be detected. The call state of *hdCall* can be any state except *idle* or *disconnected*.

dwDigitModes

Specifies the digit mode(s) that are to be monitored. A *dwDigitModes* with a value of zero cancels digit monitoring. The *dwDigitModes* parameter can have any of the following LINEDIGITMODE_ flags set:

LINEDIGITMODE_PULSE
Detect digits as audible clicks that are the result of rotary pulse sequences. Valid digits for pulse are ‘0’ through ‘9’. The **LINE_MONITORDIGITS** message should be sent as soon after the last pulse of each digit as possible.

LINEDIGITMODE_DTMF
Detect digits as DTMF tones. Valid digits for DTMF are ‘0’ through ‘9’, ‘A’, ‘B’, ‘C’, ‘D’, ‘*’, and ‘#’. The LINE_MONITORDIGITS message should be sent as soon after the onset of each tone as possible.

LINEDIGITMODE_DTMFEND
Detect and provide notification of DTMF down edges. Valid digits for DTMF are ‘0’ through ‘9’, ‘A’, ‘B’, ‘C’, ‘D’, ‘*’, and ‘#’. The **LINE_MONITORDIGITS** message should be sent as soon as possible after the end of each digit, and before any indication of the beginning of a following digit.

This function returns zero (success) when digit monitoring has been correctly initiated, not when digit monitoring has terminated. Digit monitoring remains in effect until it is explicitly disabled by a call to **TSPI_lineMonitorDigits** with *dwDigitModes* set to zero, or until the call transitions to *idle*. The service provider must terminate digit monitoring when the call goes idle. TAPI.DLL does not spontaneously call **TSPI_lineMonitorDigits** to terminate monitoring.

Although this function can be invoked in any call state, digits will typically be detected only while the call is in the *connected* state.

Each time a digit is detected, the service provider sends a **LINE_MONITORDIGITS** message to TAPI.DLL, passing the detected digit as a parameter. If both LINEDIGITMODE_DTMF and LINEDIGITMODE_DTMFEND are set in *dwDigitModes*, the two **LINE_MONITORDIGITS** messages are sent for each digit.

TAPI.DLL can use **TSPI_lineMonitorDigits** to enable or disable unbuffered digit detection. It can use **TSPI_lineGatherDigits** for buffered digit detection. After buffered digit gathering is complete, a **LINE_GATHERDIGITS** message is sent. Both buffered and unbuffered digit detection can be enabled on the same call simultaneously.

TSPI_lineMonitorMedia

[New - Windows 95]

```
LONG TSPI_lineMonitorMedia(HDRVCALL hdCall, DWORD dwMediaModes)
```

Enables and disables the detection of media modes on the specified call. When a media mode is detected, a **LINE_MONITORMEDIA** message is sent to TAPI.DLL.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

  LINEERR_INVALCALLHANDLE LINEERR_OPERATIONUNAVAIL
  LINEERR_INVALCALLSTATE LINEERR_OPERATIONFAILED
  LINEERR_INVALMEDIAMODE LINEERR_RESOURCEUNAVAIL
  LINEERR_NOMEM

hdCall

Specifies the handle to the call for which media monitoring is to be set. The call state of *hdCall* can be any state except *idle*.

dwMediaModes

Specifies the media modes to be monitored. The *dwMediaModes* parameter can have any of the following LINEMEDIAMODE_ flags set:

LINEMEDIAMODE_INTERACTIVEVOICE
One or more client applications wants to monitor the call for the presence of voice. (Note that INTERACTIVEVOICE and AUTOMATEDVOICE are equivalent for monitoring purposes.)

LINEMEDIAMODE_AUTOMATEDVOICE
One or more client applications wants to monitor the call for the presence of voice. (Note that INTERACTIVEVOICE and AUTOMATEDVOICE are equivalent for monitoring purposes.)

LINEMEDIAMODE_DATAMODEM
One or more client applications wants to monitor the call for the presence of data modem signals.

LINEMEDIAMODE_G3FAX
One or more client applications wants to monitor the call for the presence of group 3 fax signals.

LINEMEDIAMODE_TDD
One or more client applications wants to monitor the call for the presence of TDD (Telephony Devices for the Deaf) signals.

LINEMEDIAMODE_G4FAX
One or more client applications wants to monitor the call for the presence of group 4 fax signals.

LINEMEDIAMODE_DIGITALDATA
One or more client applications wants to monitor the call for the presence of unclassified digital data.

LINEMEDIAMODE_TELETEX
One or more client applications wants to monitor the call for the presence of teletex signals.

LINEMEDIAMODE_VIDEOTEX
One or more client applications wants to monitor the call for the presence of videotex signals.

LINEMEDIAMODE_TELEX
One or more client applications wants to monitor the call for the presence of telex signals.

LINEMEDIAMODE_MIXED
One or more client applications wants to monitor the call for the presence of ISDN mixed media data.

LINEMEDIAMODE_ADSI
One or more client applications wants to monitor the call for the presence of ADSI (Analog Display Services Interface) signals.

A value of zero for the *dwMediaModes* parameter cancels all media mode monitoring.

The service provider returns LINEERR_INVALMEDIAMODE if the list of media types to be monitored contains invalid information.

This function returns zero (success) when media mode monitoring has been correctly initiated, not when media mode monitoring has terminated. Media monitoring for a given media mode will remain in

effect until it is explicitly disabled by calling **TSPI_lineMonitorMedia** with a *dwMediaModes* parameter with the media mode set to zero, or until the call transitions to *idle*.

TSPI_lineMonitorMedia is primarily an event reporting mechanism. The media mode of a call, as indicated in **LINECALLINFO**, is not affected by the service provider's detection of the media mode. Only TAPI.DLL or a client application can change a call's indicated media mode using **TSPI_lineSetMediaMode**. The actual use of a particular media mode occurs through separate media stream APIs (such as Comm or WAVE).

Default media monitoring performed by the service provider for a new call appearance corresponds to the union of all media modes specified by **TSPI_lineSetDefaultMediaDetection**. Shortly after a new call is established, TAPI.DLL typically calls **TSPI_lineMonitorMedia** to reduce the set of media modes detected and reported for this call to the union of all media modes desired by the call's client applications.

It is the service provider's responsibility to cancel media monitoring when a call goes idle. It is the responsibility of TAPI.DLL to compute the union of media modes desired by all clients, and pass the result to this function. The service provider uses the set passed to this function by TAPI.DLL.

Although this function can be invoked in any call state, a call's media mode can typically only be detected while the call is certain call states. These states may be device specific. For example, in ISDN a message may indicate the media mode of the media stream before the media stream exists. Similarly, distinctive ringing or the called ID information about the call can be used to identify the media mode of a call. Otherwise, the call may have to be answered (call in the *connected* state) to allow a service provider to determine the call's media mode by filtering of the media stream. Since filtering of a call's media stream implies a computational overhead, TAPI.DLL typically uses this procedure to disable media monitoring when not required.

Because media-mode detection enabled by **TSPI_lineMonitorMedia** is implemented as a read-only operation of the call's media stream, it is not disruptive. In other words, no signals are sent on the line as a result of the setting of **TSPI_lineMonitorMedia**.

Regarding the passed media mode, TAPI.DLL guarantees that there are no reserved bits set. This means that it is the responsibility of the service provider to do any further validity checks on the media modes, such as checking whether any regular or extended media modes are indeed supported by the service provider.

TSPI_lineMonitorTones

[New - Windows 95]

```
LONG TSPI_lineMonitorTones(HDRVCALL hdCall, DWORD dwToneListID,
    LPLINEMONITORTONE const lpToneList, DWORD dwNumEntries)
```

Enables and disables the detection of inband tones on the call. Each time a specified tone is detected, a message is sent to the client application through TAPI.DLL.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALCALLSTATE	LINEERR_OPERATIONFAILED
LINEERR_INVALTONE	LINEERR_RESOURCEUNAVAIL
LINEERR_NOMEM	LINEERR_INVALPOINTER

hdCall

Specifies the handle to the call for which tone detection is to be done. The call state of *hdCall* can be any state except *idle*.

dwToneListID

Specifies the unique ID for this tone list. Several tone lists can be outstanding at once. The service provider must replace any old list having the same *dwToneListID* with the new tone list. If *lpToneList* is NULL, the tone list with *dwToneListID* is simply dropped. In any case, other tone lists with different *dwToneListID*s are kept unchanged.

lpToneList

Specifies a list of tones to be monitored, of type **LINEMONITORTONE**. Each tone in this list has an application-defined tag field that is used to identify individual tones in the list for the purpose of reporting a tone detection. Tone monitoring in progress is canceled or changed by calling this operation with either NULL for *lpToneList* or with another tone list. The service provider must copy the tone list into its own memory for later reference, rather than simply retaining the pointer into application memory.

dwNumEntries

Specifies the number of entries in *lpToneList*. The *dwNumEntries* parameter is ignored if *lpToneList* is NULL. Note that this parameter is not validated by TAPI.DLL when this function is called.

This function returns zero (success) when tone monitoring has been correctly initiated; not when tone monitoring has terminated. As with media monitoring, tone monitoring remains in effect for a given tone list until that tone list is explicitly disabled—by calling **TSPI_lineMonitorTones** with the same *dwToneListID* and another tone list (or a NULL tone list), or until the call transitions to *idle*.

Although this function can be invoked in any call state except *idle*, tones can typically only be detected while the call is in the *connected* state. Tone detection usually requires computational resources. Depending on the service provider and other activities that compete for such resources, the number of tones that can be detected may vary over time. Also, an equivalent amount of resources may be consumed for monitoring a single triple frequency tone versus three single frequency tones. If resources are overcommitted, the service provider should return LINEERR_RESOURCEUNAVAIL.

The service provider monitors for all tones in all tone lists concurrently. When a tone is detected, each matching tone from each tone list is reported separately using a LINE_MONITORTONE message. Each tone report includes both the tone list ID and the application-specific tag. Some service providers may not be able to discriminate very close tones, so that multiple matches may be reported even for tones whose descriptions are not strictly identical.

Note that **TSPI_lineMonitorTones** is also used to detect silence. Silence is specified as a tone with all zero frequencies.

The corresponding function at the TAPI interface does not include a *dwToneListID* parameter. The inclusion of this parameter at the TSPI interface allows TAPI.DLL to forward the union of all tone monitoring lists from all applications to the service provider, while still retaining the ability to filter and forward the tone detection events according to application. This gives service-provider designers the maximum flexibility to determine the degree to which they can discriminate very close tones, since TAPI.DLL makes no assumptions about what tone descriptions are considered identical.

TSPI_lineNegotiateExtVersion

[New - Windows 95]

```
LONG TSPI_lineNegotiateExtVersion(DWORD dwDeviceID, DWORD dwTSPIVersion,
    DWORD dwLowVersion, DWORD dwHighVersion, LPDWORD lpdwExtVersion)
```

Returns the highest extension version number the service provider is willing to operate under for this device given the range of possible extension versions.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INCOMPATIBLEAPIVERSION	LINEERR_OPERATIONUNAVAIL
LINEERR_INCOMPATIBLEEXTVERSION	LINEERR_OPERATIONFAILED
LINEERR_NODRIVER	LINEERR_RESOURCEUNAVAIL
LINEERR_NOMEM	

dwDeviceID

Identifies the line device for which interface version negotiation is to be performed. The value INITIALIZE_NEGOTIATION may not be used for this function.

dwTSPIVersion

Specifies an interface version number that has already been negotiated for this device using **TSPI_lineNegotiateTSPIVersion**. This function operates according to the interface specification at this version level.

dwLowVersion

Specifies the lowest extension version number under which TAPI.DLL or its client application is willing to operate. The most-significant WORD is the major version number and the least-significant WORD is the minor version number. Note that this parameter is not validated by TAPI.DLL when this function is called.

dwHighVersion

Specifies the highest extension version number under which TAPI.DLL or its client application is willing to operate. The most-significant WORD is the major version number and the least-significant WORD is the minor version number. Note that this parameter is not validated by TAPI.DLL when this function is called.

lpdwExtVersion

Specifies a far pointer to a DWORD. Upon a successful return from this function, the service provider fills this location with the highest extension version number, within the range requested by the caller, under which the service provider is willing to operate. The most-significant WORD is the major version number and the least-significant WORD is the minor version number. If the requested range does not overlap the range supported by the service provider, the function returns LINEERR_INCOMPATIBLEEXTVERSION.

This function may be called before or after the device has been opened by TAPI.DLL. If the device is currently open and has an extension version selected, the function should give that version number if it is within the requested range. If the selected version number is outside the requested range, the function returns LINEERR_INCOMPATIBLEEXTVERSION.

TSPI_lineNegotiateTSPIVersion

[New - Windows 95]

```
LONG TSPI_lineNegotiateTSPIVersion(DWORD dwDeviceID, DWORD dwLowVersion,
    DWORD dwHighVersion, LPDWORD lpdwTSPIVersion)
```

Returns the highest SPI version the service provider is willing to operate under for this device given the range of possible SPI versions.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INCOMPATIBLEAPIVERSION	LINEERR_OPERATIONUNAVAIL
LINEERR_NODRIVER	LINEERR_OPERATIONFAILED
LINEERR_NOMEM	LINEERR_RESOURCEUNAVAIL

dwDeviceID

Identifies the line device for which interface version negotiation is to be performed. In addition to device IDs within the range the service provider supports, this may be the value:

INITIALIZE_NEGOTIATION
: This value is used to signify that an overall interface version is to be negotiated.

dwLowVersion

Specifies the lowest TSPI version number under which TAPI.DLL is willing to operate. The most-significant WORD is the major version number and the least-significant WORD is the minor version number.

dwHighVersion

Specifies the highest TSPI version number under which TAPI.DLL is willing to operate. The most-significant WORD is the major version number and the least-significant WORD is the minor version number.

lpdwTSPIVersion

Specifies a far pointer to a DWORD. The service provider fills this location with the highest TSPI version number, within the range requested by the caller, under which the service provider is willing to operate. The most-significant WORD is the major version number and the least-significant WORD is the minor version number. If the requested range does not overlap the range supported by the service provider, the function returns LINEERR_INCOMPATIBLEAPIVERSION.

Note that when *dwDeviceID* is INITIALIZE_NEGOTIATION, this function must not return LINEERR_OPERATIONUNAVAIL, since this function (with that value) is mandatory for negotiating the overall interface version even if the service provider supports no line devices.

TSPI_lineOpen

[New - Windows 95]

```
LONG TSPI_lineOpen(DWORD dwDeviceID, HTAPILINE htLine,
    LPHDRVLINE lphdLine, DWORD dwTSPIVersion, LINEEVENT lpfnEventProc)
```

Opens the line device whose device ID is given, returning the service provider's handle for the device. The service provider must retain TAPI.DLL's handle for the device for use in subsequent calls to the **LINEEVENT** callback procedure.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_ALLOCATED	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALMEDIAMODE	LINEERR_OPERATIONFAILED
LINEERR_NODRIVER	LINEERR_RESOURCEUNAVAIL
LINEERR_NOMEM	

dwDeviceID

Identifies the line device to be opened.

htLine

Specifies TAPI.DLL's handle for the line device to be used in subsequent calls to the **LINEEVENT** callback procedure to identify the device.

lphdLine

A far pointer to a HDRVLINE where the service provider fills in its handle for the line device.

dwTSPIVersion

The TSPI version.

lpfnEventProc

A far pointer to the **LINEEVENT** callback procedure supplied by TAPI.DLL that the service provider will call to report subsequent events on the line.

The service provider should reserve any non-sharable resources that are required to manage the line. However, any actions which can be postponed to **lineMakeCall** should be. It is a design assumption in TAPI that **lineOpen** is an "inexpensive" operation. For example, if the line is opened in monitor mode only, it should not be necessary for a COMM-port-based service provider to open the COMM port.

This procedure does not correspond directly to any procedure at the TAPI level, at which the functions of enabling device-specific extensions, selecting line characteristics, and setting media mode detection are included in the functionality defined by **lineOpen**. At the TSPI level, these additional capabilities are separated out into **TSPI_lineNegotiateExtVersion**, **TSPI_lineSetDefaultMediaDetection** and **TSPI_lineConditionalMediaDetection**.

TSPI_linePark

[New - Windows 95]

```
LONG TSPI_linePark(DRV_REQUESTID dwRequestID, HDRVCALL hdCall,
    DWORD dwParkMode, LPCSTR lpszDirAddress,
    LPVARSTRING lpNonDirAddress)
```

Parks the specified call according to the specified park mode.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_NOMEM
LINEERR_INVALPARKMODE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALCALLSTATE	LINEERR_OPERATIONFAILED
LINEERR_INVALADDRESS	LINEERR_RESOURCEUNAVAIL

dwRequestID

Specifies the identifier of the asynchronous request.

hdCall

Specifies the handle to the call to be parked. The call state of *hdCall* can be *connected*.

dwParkMode

Specifies the park mode with which the call is to be parked. Only one of the following LINEPARKMODE_ flags may be set at a time:

LINEPARKMODE_DIRECTED

TAPI.DLL specifies the address at which address the call is to be parked in *lpszDirAddress*.

LINEPARKMODE_NONDIRECTED

This operation reports to TAPI.DLL where the call has been parked in *lpNonDirAddress*.

lpszDirAddress

Specifies a far pointer to NULL-terminated ASCII string that indicates the address where the call is to be parked when using directed park. The address is in dialable address format. This parameter is ignored for nondirected park.

lpNonDirAddress

Specifies a far pointer to a structure of type **VARSTRING**. For nondirected park, the address where the call is parked is returned in this structure. This parameter is ignored for directed park. Within the **VARSTRING** structure, *dwStringFormat* must be set to **STRINGFORMAT_ASCII** (an ASCII string buffer containing a NULL-terminated string), and the terminating NULL is accounted for in the *dwStringSize*.

All fields of the **VARSTRING** structure, except **dwTotalSize**, are filled in by the service provider. **dwTotalSize** is filled in by TAPI.DLL, and the service provider must *not* overwrite this value.

Under directed park, the client application (through TAPI.DLL) specifies the address at which it wants to park the call. Under nondirected park, the switch determines the address and provides this to TAPI.DLL. In either case, a parked call can be unparked by specifying this address.

The parked call typically enters the *idle* call state after it has been successfully parked. The service provider reports the new state using a **LINE_CALLSTATE** message. A subsequent **TSPI_lineUnpark** creates a new, distinct call handle, regardless of whether **TSPI_lineCloseCall** has destroyed the old handle.

Some switches may remind the user after a call has been parked for some long amount of time. The service provider would report this to TAPI.DL as an *offering* call with a call reason set to *reminder* (if this is known).

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

TSPI_linePickup

[New - Windows 95]

```
LONG TSPI_linePickup(DRV_REQUESTID dwRequestID, HDRVLINE hdLine,
    DWORD dwAddressID, HTAPICALL htCall, LPHDRVCALL lphdCall,
    LPCSTR lpszDestAddress, LPCSTR lpszGroupID)
```

Picks up a call alerting at the specified destination address and returns a call handle for the picked up call. If invoked with NULL for the *lpszDestAddress* parameter, a group pickup is performed. If required by the device capabilities, *lpszGroupID* specifies the group ID to which the alerting station belongs.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALLINEHANDLE	LINEERR_NOMEM
LINEERR_INVALADDRESSID	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALADDRESS	LINEERR_OPERATIONFAILED
LINEERR_INVALGROUPID	LINEERR_RESOURCEUNAVAIL

dwRequestID

Specifies the identifier of the asynchronous request.

hdLine

Specifies the handle to the line on which a call is to be picked up.

dwAddressID

Specifies the address on *hdLine* at which the pickup is to be originated.

htCall

Specifies TAPI.DLL's handle to the new call. The service provider must save this and use it in all subsequent calls to the **LINEEVENT** procedure reporting events on the call.

lphdCall

Specifies a far pointer to an HDRVCALL representing the service provider's identifier for the call. The service provider must fill this location with its handle for the call before this procedure returns. This handle is ignored by TAPI.DLL if the function results in an error.

lpszDestAddress

Specifies a far pointer to a NULL terminated ASCII string that contains the address whose call is to be picked up. The address is standard link format.

lpszGroupID

Specifies a far pointer to a NULL-terminated ASCII string containing the group ID to which the alerting station belongs. This parameter is required on some switches to pick up calls outside of the current pickup group.

Note that *lpszGroupID* can be specified by itself with a NULL pointer for *lpszDestAddress*. Alternatively, *lpszGroupID* can be specified in addition to *lpszDestAddress*, if required by the device. It can also be NULL itself.

When a call has been picked up successfully, the service provider notifies TAPI.DLL with the **LINE_CALLSTATE** message about call state changes. The **LINECALLINFO** structure supplies information about the call that was picked up. It will list the reason for the call as *pickup*. This structure is available by calling **TSPI_lineGetCallInfo**.

The service provider should set LINEADDRCAPFLAGS_PICKUPCALLWAIT to TRUE in the **LINEADDRESSCAPS** structure if **TSPI_linePickup** can be used to pick up a call for which the user has audibly detected the call-waiting signal but for which the provider is unable to perform the detection. This gives the user a mechanism to "answer" a waiting call even though the service provider was unable to detect the call-waiting signal. When TSPI_linePickup is being used to pick up a call-waiting call, both *lpszDestAddress* and *lpszGroupID* pointer parameters will be NULL. The service provider creates a new call handle for the waiting call and passes that handle to the user in *lphdCall*. *dwAddressID* will most often be zero (particularly in single-line residential cases).

Once **TSPI_linePickup** has been used to pick up the second call, **TSPI_lineSwapHold** can be used to toggle

between them. **TSPI_lineDrop** can be used to drop one (and toggle to the other), and so forth. If the user wants to drop the current call and pick up the second call, they will call **TSPI_lineDrop** when they get the call-waiting beep, wait for the second call to ring, and then call **TSPI_lineAnswer** on the new call handle. The service provider sets the LINEADDRFEATURE_PICKUP flag in the **dwAddressFeatures** field in **LINEADDRESSSTATUS** to indicate when pickup is actually possible.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

This function differs from the corresponding TAPI function in that it follows the TSPI model for beginning the lifetime of a call. TAPI.DLL and the service provider exchange opaque handles representing the call with one another. In addition, the service provider is permitted to do callbacks for the new call before it returns from this procedure. In any case, the service provider must also treat the handle it returned as “not yet valid” until after the matching **ASYNC_COMPLETION** message reports success. In other words, it must not issue any **LINEEVENT** messages for the new call or include it in call counts in messages or status data structures for the line.

TSPI_linePrepareAddToConference

[New - Windows 95]

```
LONG TSPI_linePrepareAddToConference(DRV_REQUESTID dwRequestID,
    HDRVCALL hdConfCall, HTAPICALL htConsultCall,
    LPHDRVCALL lphdConsultCall, LPLINECALLPARAMS const lpCallParams)
```

Prepares an existing conference call for the addition of another party. It creates a new, temporary consultation call. The new consultation call can be subsequently added to the conference call.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_BEARERMODEUNAVAIL	LINEERR_INVALLINESTATE
LINEERR_CALLUNAVAIL	LINEERR_INVALMEDIAMODE
LINEERR_CONFERENCEFULL	LINEERR_INVALRATE
LINEERR_INUSE	LINEERR_NOMEM
LINEERR_INVALADDRESSMODE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALBEARERMODE	LINEERR_OPERATIONFAILED
LINEERR_INVALCALLPARAMS	LINEERR_RATEUNAVAIL
LINEERR_INVALCALLSTATE	LINEERR_RESOURCEUNAVAIL
LINEERR_INVALCONFCALLHANDLE	LINEERR_USERUSERINFOTOOBIG

dwRequestID

Specifies the identifier of the asynchronous request.

hdConfCall

Specifies the handle to a conference call. The call state of *hdConfCall* can be *connected*.

htConsultCall

Specifies TAPI.DLL's handle to the new, temporary consultation call. The service provider must save this and use it in all subsequent calls to the **LINEEVENT** procedure reporting events on the new call. The call state of *hdAddCall* is not applicable.

lphdConsultCall

Specifies a far pointer to an HDRVCALL representing the service provider's identifier for the new, temporary consultation call. The service provider must fill this location with its handle for the new call before this procedure returns. This handle is invalid if the function results in an error.

lpCallParams

Specifies a far pointer to call parameters to be used when establishing the consultation call. This parameter will be set to NULL if no special call setup parameters are desired.

The service provider returns LINEERR_INVALLINESTATE if the line is currently not in a state in which this operation can be performed. A list of currently valid operations must be indicated by the service provider in the **dwLineFeatures** field (of the type **LINEFEATURE**) in the **LINEDEVSTATUS** structure.

The service provider returns LINEERR_INVALCALLSTATE if the conference call is not in a valid state for the requested operation.

This function places an existing conference call in the *onHoldPendingConference* state and creates a consultation call that can be added later to the existing conference call with **TSPI_lineAddToConference**.

The consultation call can be canceled using **TSPI_lineDrop**. It may also be possible for TAPI.DLL to swap between the consultation call and the held conference call with **TSPI_lineSwapHold**. The service provider initially does media monitoring on the new call for at least the set of media modes that were monitored for on the line.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck

pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

This function differs from the corresponding TAPI function in that it follows the TSPI model for beginning the lifetime of a call. TAPI.DLL and the service provider exchange opaque handles representing the call with one another. In addition, the service provider is permitted to do callbacks for the new call before it returns from this procedure. In any case, the service provider must also treat the handle it returned as “not yet valid” until after the matching **ASYNC_COMPLETION** message reports success. In other words, it must not issue any **LINEEVENT** messages for the new call or include it in call counts in messages or status data structures for the line.

TSPI_lineRedirect

[New - Windows 95]

```
LONG TSPI_lineRedirect(DRV_REQUESTID dwRequestID, HDRVCALL hdCall,
    LPCSTR lpszDestAddress, DWORD dwCountryCode)
```

Redirects the specified offering call to the specified destination address.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_NOMEM
LINEERR_INVALCALLSTATE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALCOUNTRYCODE	LINEERR_OPERATIONFAILED
LINEERR_INVALADDRESS	LINEERR_RESOURCEUNAVAIL

dwRequestID

Specifies the identifier of the asynchronous request.

hdCall

Specifies the handle to the call to be redirected. The call state of *hdCall* can be *offering*.

lpszDestAddress

Specifies a far pointer to the destination address. This follows the standard link format.

dwCountryCode

Specifies the country code of the party the call is redirected to. If a value of zero is specified, a default will be used by the implementation. Note that this parameter is not validated by TAPI.DLL when this function is called.

The service provider does not redirect the call if it returns LINEERR_INVALADDRESS.

When this function is invoked, the service provider deflects the offering call to another address without first answering the call. Call redirect differs from call forwarding in that call forwarding is performed by the switch without the involvement of the called station; redirection can be done on a call-by-call basis by a client application, for example driven by caller ID information. It differs from call transfer in that transferring a call requires the call first be answered.

After a call has been successfully redirected, the call will typically transition to *idle*. The service provider indicates the new state using a **LINE_CALLSTATE** message.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

TSPI_lineRemoveFromConference

[New - Windows 95]

```
LONG TSPI_lineRemoveFromConference(DRV_REQUESTID dwRequestID,
    HDRVCALL hdCall)
```

Removes the specified call from the conference call to which it currently belongs. The remaining calls in the conference call are unaffected.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALCALLSTATE	LINEERR_OPERATIONFAILED
LINEERR_NOMEM	LINEERR_RESOURCEUNAVAIL

dwRequestID

Specifies the identifier of the asynchronous request.

hdCall

Specifies the handle to the call to be removed from the conference. The call state of *hdCall* can be *conferenced*.

This operation removes a party that currently belongs to a conference call. After the call has been successfully removed, it may be possible to further manipulate it using its handle. The availability of this operation and its result are likely to be limited in many implementations. For example, in many implementations, only the most recently added party may be removed from a conference, and the removed call may be automatically dropped (becomes *idle*). The service provider indicates its capabilities in **LINEDEVCAPS** with regard to the available effects of removing a call from a conference.

If the removal of a participant from a conference is supported, the service provider must indicate in the **dwRemoveFromConfState** field in **LINEADDRESSCAPS** the callstate to which the call will transition after it is removed from the conference.

TSPI_lineSecureCall

[New - Windows 95]

```
LONG TSPI_lineSecureCall(DRV_REQUESTID dwRequestID, HDRVCALL hdCall)
```

Secures the call from any interruptions or interference that may affect the call's media stream.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALCALLSTATE	LINEERR_OPERATIONFAILED
LINEERR_NOMEM	LINEERR_RESOURCEUNAVAIL

dwRequestID

Specifies the identifier of the asynchronous request.

hdCall

Specifies the handle to the call to be secured. The call state of *hdCall* can be any state except *idle*.

A call can be secured to avoid interference. For example, in an analog environment, call waiting tones may destroy a fax or modem session on the original call. **TSPI_lineSecureCall** allows an existing call to be secured, **TSPI_lineMakeCall** provides the option to secure the call from the time of call setup. The securing of a call remains in effect for the duration of the call.

TSPI_lineSelectExtVersion

[New - Windows 95]

```
LONG TSPI_lineSelectExtVersion(HDRVLINE hdLine, DWORD dwExtVersion)
```

Selects the indicated extension version for the indicated line device. Subsequent requests operate according to that extension version.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INCOMPATIBLEEXTVERSION	LINEERR_OPERATIONFAILED
LINEERR_NOMEM	LINEERR_RESOURCEUNAVAIL
LINEERR_OPERATIONUNAVAIL	

hdLine

Specifies the handle to the line for which an extension version is to be selected.

dwExtVersion

Specifies the extension version to be selected. This version number has been negotiated using **TSPI_lineNegotiateExtVersion**. The most significant WORD is the major version number and the least significant WORD is the minor version number. Calling this function with a *dwExtVersion* of zero cancels the current selection.

The service provider selects the indicated extension version. Although the indicated version number may have been successfully negotiated, a different extension version may have been selected in the interim, in which case this function fails (returning LINEERR_INCOMPATIBLEEXTVERSION).

Subsequent operations on the line after an extension version has been selected behave according to that extension version. Subsequent attempts to negotiate the extension version report strictly the selected version or 0 (if the requested range does not include the selected version). Calling this procedure with the special extension version 0 cancels the current selection. The device becomes once again capable of supporting its full range of extension version numbers.

This function has no direct correspondence at the TAPI level, where selecting an extension version is bundled with the other functions of **lineOpen**. The **TSPI_lineSelectExtVersion** function is typically called in two situations: (1) An application requested to open a line, the resulting change of media mode monitoring was successful, the application requested that a particular extension version be used, and no extension version was currently selected. (2) The last application using a particular extension version closed the line, and the extension version selection can be cancelled.

TSPI_lineSendUserUserInfo

[New - Windows 95]

```
LONG TSPI_lineSendUserUserInfo(DRV_REQUESTID dwRequestID,
    HDRVCALL hdCall, LPCSTR lpsUserUserInfo, DWORD dwSize)
```

Sends user-to-user information to the remote party on the specified call.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_OPERATIONFAILED
LINEERR_INVALCALLSTATE	LINEERR_RESOURCEUNAVAIL
LINEERR_NOMEM	LINEERR_USERUSERINFOTOOBIG
LINEERR_OPERATIONUNAVAIL	

dwRequestID

Specifies the identifier of the asynchronous request.

hdCall

Specifies the handle to the call on which to send user-to-user information. The call state of *hdCall* can be *connected*.

lpsUserUserInfo

Specifies a far pointer to a string containing user-to-user information to be sent to the remote party. User-to-user information is only sent if supported by the underlying network (see **LINEDEVCAPS**).

dwSize

Specifies the size in bytes of the user-to-user information in *lpsUserUserInfo*.

This function can be used to send user-to-user information at any time during a connected call. If the size of the specified information to be sent is larger than what may fit into a single network message (as in ISDN), the service provider is responsible for breaking the information up into a sequence of chained network messages (using “more data”).

User-to-user information can also be sent as part of call accept, call reject, call redirect, and when making calls. User-to-user information can also be received. The received information is reported in the call’s **LINECALLINFO** structure. Whenever user-to-user information arrives after call offering or prior to call disconnect, a **LINE_CALLINFO** message with a *UserUserInfo* parameter will notify TAPI.DLL that user-to-user information in the call-information record has changed. If multiple network messages are chained, the information is assembled by the service provider and a single message is sent to TAPI.DLL.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

TSPI_lineSetAppSpecific

[New - Windows 95]

```
LONG TSPI_lineSetAppSpecific(HDRVCALL hdCall, DWORD dwAppSpecific)
```

Sets the application-specific field of the specified call's **LINECALLINFO** structure.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

  LINEERR_INVALCALLHANDLE LINEERR_OPERATIONFAILED
  LINEERR_NOMEM LINEERR_RESOURCEUNAVAIL
  LINEERR_OPERATIONUNAVAIL

hdCall

Specifies the handle to the call whose application-specific field is to be set. The call state of *hdCall* can be any state.

dwAppSpecific

Specifies the new content of the **dwAppSpecific** field for the call's **LINECALLINFO** structure. This value is uninterpreted by the service provider. Note that this parameter is not validated by TAPI.DLL when this function is called.

The application-specific field in the **LINECALLINFO** data structure that exists for each call is uninterpreted by the Telephony API or any of its service providers. Its usage is entirely defined by the applications. The field can be read from the **LINECALLINFO** record returned by **TSPI_lineGetCallInfo**. However, **TSPI_lineSetAppSpecific** must be used to set the field so that changes become visible to other applications. When this field is changed, the service provider sends a **LINE_CALLINFO** message with an indication that the **AppSpecific** field has changed.

TSPI_lineSetCallParams

[New - Windows 95]

```
LONG TSPI_lineSetCallParams(DRV_REQUESTID dwRequestID, HDRVCALL hdCall,
    DWORD dwBearerMode, DWORD dwMinRate, DWORD dwMaxRate,
    LPLINEDIALPARAMS const lpDialParams)
```

Sets certain parameters for an existing call.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_RATEUNAVAIL
LINEERR_INVALCALLSTATE	LINEERR_NOMEM
LINEERR_INVALBEARERMODE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALPOINTER	LINEERR_OPERATIONFAILED
LINEERR_INVALRATE	LINEERR_RESOURCEUNAVAIL
LINEERR_BEARERMODEUNAVAIL	

dwRequestID

Specifies the identifier of the asynchronous request.

hdCall

Specifies the handle to the call whose parameters are to be changed. The call state can be any state except *idle* and *disconnected*.

dwBearerMode

Specifies the new bearer mode for the call. The *dwBearerMode* parameter can have only one of the following LINEBEARERMODE_flags set:

LINEBEARERMODE_VOICE
: This is a regular 3.1kHz analog voice grade bearer service. Bit integrity is not assured. Voice can support fax and modem media modes.

LINEBEARERMODE_SPEECH
: This corresponds to G.711 speech transmission on the call. The network may use processing techniques such as analog transmission, echo cancellation and compression/decompression. Bit integrity is not assured. Speech is not intended to support fax and modem media modes.

LINEBEARERMODE_MULTIUSE
: The multi-use mode defined by ISDN.

LINEBEARERMODE_DATA
: The unrestricted data transfer on the call. The data rate is specified separately.

LINEBEARERMODE_ALTSPEECHDATA
: The alternate transfer of speech or unrestricted data on the same call (ISDN).

LINEBEARERMODE_NONCALLSIGNALING
: This corresponds to a non-call-associated signaling connection from the application to the service provider or switch (treated as a “media stream” by TAPI.DLL).

dwMinRate

Specifies a lower bound for the call’s new data rate. TAPI.DLL is willing to accept a new rate as low as this one. Note that this parameter is not validated by TAPI.DLL when this function is called.

dwMaxRate

Specifies an upper bound for the call’s new data rate. This is the maximum data rate TAPI.DLL would like. Equal values for *dwMinRate* and *dwMaxRate* indicate that an exact data rate is required. Note that this parameter is not validated by TAPI.DLL when this function is called.

lpDialParams

Specifies a far pointer to the new dial parameters for the call, of type **LINEDIALPARAMS**. If this parameter is NULL, it indicates that the call’s current dialing parameters are to be used.

This operation is used to change the parameters of an existing call. Examples of its usage include changing the bearer mode and/or the data rate of an existing call.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

TSPI_lineSetDefaultMediaDetection

[New - Windows 95]

```
LONG TSPI_lineSetDefaultMediaDetection(HDRVLINE hdLine,
    DWORD dwMediaModes)
```

This procedure tells the service provider the new set of media modes to detect for the indicated line (replacing any previous set). It also sets the initial set of media modes that should be monitored for on subsequent calls (inbound or outbound) on this line.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

  LINEERR_INVALLINEHANDLE LINEERR_OPERATIONFAILED
  LINEERR_INVALMEDIAMODE LINEERR_RESOURCEUNAVAIL
  LINEERR_NOMEM LINEERR_NODRIVER
  LINEERR_OPERATIONUNAVAIL

hdLine

Specifies the handle to the line to have media monitoring set.

dwMediaModes

Specifies the media mode(s) of interest to TAPI.DLL. This parameter uses the following LINEMEDIAMODE_ constants:

LINEMEDIAMODE_UNKNOWN
TAPI.DLL wants to handle calls of unknown media type (unclassified calls).

LINEMEDIAMODE_INTERACTIVEVOICE
TAPI.DLL wants to handle calls of the interactive voice media type. That is, it manages live voice calls.

LINEMEDIAMODE_AUTOMATEDVOICE
TAPI.DLL wants to handle calls in which there is voice data managed by an automated application. In cases where the service provider does not distinguish between automated or interactive voice, this should be treated the same as interactive voice.

LINEMEDIAMODE_DATAMODEM
TAPI.DLL wants to handle calls with the data modem media mode.

LINEMEDIAMODE_G3FAX
TAPI.DLL wants to handle calls of the group 3 fax media type.

LINEMEDIAMODE_TDD
TAPI.DLL wants to handle calls with the TDD (Telephony Devices for the Deaf) media mode.

LINEMEDIAMODE_G4FAX
TAPI.DLL wants to handle calls of the group 4 fax media type.

LINEMEDIAMODE_DIGITALDATA
TAPI.DLL wants to handle calls of the digital data media type.

LINEMEDIAMODE_TELETEX
TAPI.DLL wants to handle calls with the teletex media mode.

LINEMEDIAMODE_VIDEOTEX
TAPI.DLL wants to handle calls with the videotex media mode.

LINEMEDIAMODE_TELEX
TAPI.DLL wants to handle calls with the telex media mode.

LINEMEDIAMODE_MIXED
TAPI.DLL wants to handle calls with the ISDN mixed media mode.

LINEMEDIAMODE_ADSI
TAPI.DLL wants to handle calls with the ADSI (Analog Display Services Interface) media mode.

TAPI.DLL typically calls this function to update the set of detected media modes for the line to the union of all modes selected by all outstanding lineOpens whenever a line is Opened or Closed at the TAPI level. A **lineOpen** call attempt is rejected if media detection is rejected. A single call to this procedure is typically the result of a **lineOpen** call that does not specify the device ID LINEMAPPER. The Device ID LINEMAPPER is never used at the TSPI level.

It is the responsibility of TAPI.DLL to compute the union of media modes desired by all clients and pass the

result to this function. The service provider uses the set passed to this function by TAPI.DLL. TAPI.DLL insures that the *dwMediaModes* parameter has at least one bit set and that no reserved bits are set. It is the service provider's responsibility to do any further validity checks on the media modes, such as checking whether any regular or extended media modes are indeed supported by the service provider. The union of all media modes may be the value 0 if the applications which have the line open are all either monitors or not interested in handling incoming calls.

There is no directly corresponding function at the TAPI level. This procedure corresponds to the "request media modes" implied for the specific line by the **lineOpen** procedure when it is called with the specific device ID (other than LINEMAPPER).

TSPI_lineSetDevConfig

[New - Windows 95]

```
LONG TSPI_lineSetDevConfig (DWORD dwDeviceID,
    LPVOID const lpDeviceConfig, DWORD dwSize, LPCSTR lpszDeviceClass)
```

Restores the configuration of a device associated one-to-one with the line device from an "" data structure previously obtained using **TSPI_lineGetDevConfig**. The contents of this data structure are specific to the line [service provider] and device class.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are:

  LINEERR_INVALDEVICECLASS LINEERR_NOMEM
  LINEERR_INVALPOINTER LINEERR_OPERATIONUNAVAIL
  LINEERR_INVALPARAM LINEERR_OPERATIONFAILED
  LINEERR_INVALLINESTATE LINEERR_RESOURCEUNAVAIL
  LINEERR_NODRIVER

dwDeviceID

Specifies the line device to be configured.

lpDeviceConfig

Specifies a far pointer to the configuration data structure which had been returned in the variable portion of the **VARSTRING** structure by **TSPI_lineGetDevConfig**.

dwSize

Specifies the number of bytes in the structure pointed to be *lpDeviceConfig*. This value will have been returned in the **dwStringSize** field in the **VARSTRING** structure returned by **TSPI_lineGetDevConfig**.

lpszDeviceClass

Specifies a far pointer to a NULL-terminated ASCII string that specifies the device class of the device whose configuration is to be restored. Valid device class strings are the same as those specified for the **TSPI_lineGetID** function when it is applied to a "line" device (that is, when *dwSelect* has the value LINECALLSELECT_LINE).

The call state is device specific.

The service provider returns LINEERR_INVALPARAM if the information contained in the structure pointed to by *lpDeviceConfig* is not valid for this device.

The service provider returns LINEERR_INVALLINESTATE if the device configuration may not be changed in the current line state. The line may be in use by another application.

This function can be used to restore the configuration of a device associated one-to-one with the line device from a data structure previously retrieved from the service provider using the **TSPI_lineGetDevConfig** function. The *lpszDeviceClass* parameter selects which of among possibly several different classes of devices is to have its configuration restored. The set of supported classes is restricted to those whose devices correspond one-to-one with the line device.

A service provider should typically allow the "tapi/line" device class under this function. It would restore parameters that have "line" scope, such as the list of addresses in this line and the list of physical hardware devices such as COMM ports corresponding to the addresses or the maximum number of concurrent calls (if configurable).

In general, this function would NOT allow media-related device classes such as mci waveaudio, low level wave, or datamodem device classes, since these usually apply to a particular call or a particular address. Since there may be more than one of these per line device, the identification of the particular call or address simply by the line device ID parameter in this function would be ambiguous. An exception can be made for call-specific or address-specific device classes in cases where there is class configuration information that applies to the entire line device scope, such as initial defaults.

There are several reasons why "exceptional" support for call-specific and address-specific device classes is of only limited value under this function. First, since these classes may be ambiguous on multiple-address, multiple-call service providers, only a subset of service providers will support them. Application programs are not likely to

add a device-specific dependency on the inclusion of these classes in this function. Second, as higher-level media "classes" emerge that implement high-level protocols such as dial-in file system access in terms of low-level transport APIs, configuration for these classes will tend toward "instance" scope instead of "class" scope. The high-level media API will have to supply its own functions to configure call-specific or address-specific instances.

Whatever sort of devices and device classes this function supports, it can potentially affect two kinds of configuration information: permanent and temporary. Permanent information survives across different "opens" of the line, and even across different "inits" of the service provider itself. Temporary information survives only within a unique "open" of the line. When the line is closed, any such temporary information that has been set or retrieved through **TSPI_lineGetDevConfig** may revert to default or undefined values. The caller can reliably retrieve any temporary configuration only by a sequence such as **TSPI_lineOpen**, **TSPI_lineConfigDialog**, **TSPI_lineGetDevConfig**. The caller can reliably set temporary configuration information retrieved by such a sequence through a sequence such as **TSPI_lineOpen**, **TSPI_lineSetDevConfig**. The temporary part of configuration remains stable only until the next **TSPI_lineConfigDialog**, **TSPI_lineSetDevConfig**, or **TSPI_lineClose**. The service provider must take care of storing any permanent part of the configuration, typically in an "INI" file, and reloading it whenever the service provider is initialized.

The exact format of the data contained within the structure passed to this function is specific to the line and device class API, is undocumented, and is undefined. The structure passed to this function cannot be directly accessed or manipulated by the application, but can only be stored intact and later used from a previous **TSPI_lineGetDevConfig** to obtain the settings. The structure also cannot necessarily be passed to other devices, even of the same device class (although this may work in some instances, it is not guaranteed). A service provider should check this data structure for consistency to guard against failures due to a client application passing incompatible information.

Note that some service providers may permit the configuration to be set while a device is in use, and others may not.

TSPI_lineSetMediaControl

[New - Windows 95]

```
LONG TSPI_lineSetMediaControl(HDRVLINE hdLine, DWORD dwAddressID,
    HDRVCALL hdCall, DWORD dwSelect,
    LPLINEMEDIACONTROLDIGIT const lpDigitList, DWORD dwDigitNumEntries,
    LPLINEMEDIACONTROLMEDIA const lpMediaList, DWORD dwMediaNumEntries,
    LPLINEMEDIACONTROLTONE const lpToneList, DWORD dwToneNumEntries,
    LPLINEMEDIACONTROLCALLSTATE const lpCallStateList,
    DWORD dwCallStateNumEntries)
```

Enables and disables control actions on the media stream associated with the specified line, address, or call. Media control actions can be triggered by the detection of specified digits, media modes, custom tones, and call states. The new specified media controls replace all the ones that were in effect for this line, address, or call prior to this request.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALADDRESSID	LINEERR_INVALPOINTER
LINEERR_INVALCALLHANDLE	LINEERR_INVALTONELIST
LINEERR_INVALCALLSELECT	LINEERR_NOMEM
LINEERR_INVALCALLSTATELIST	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALDIGITLIST	LINEERR_OPERATIONFAILED
LINEERR_INVALLINEHANDLE	LINEERR_RESOURCEUNAVAIL
LINEERR_INVALMEDIALIST	

hdLine

Specifies the handle to a line.

dwAddressID

Specifies an address on the given open line device. Note that this parameter is not validated by TAPI.DLL when this function is called.

hdCall

Specifies the handle to a call. The call state of *hdCall* can be any state.

dwSelect

Specifies whether media control is requested is associated with a single call, is the default for all calls on an address, or is the default for all calls on a line. This parameter uses the following LINECALLSELECT_ constants:

LINECALLSELECT_LINE
: Selects the specified line device. The *hdLine* parameter must be a valid line handle; *hdCall* and *dwAddressID* must be ignored.

LINECALLSELECT_ADDRESS
: Selects the specified address on the line. Both *hdLine* and *dwAddressID* must be valid; *hdCall* must be ignored.

LINECALLSELECT_CALL
: Selects the specified call. *hdCall* must be valid; *hdLine* and *dwAddressID* must be ignored.

lpDigitList

Specifies a far pointer to the array that contains the digits that are to trigger media control actions, of type **LINEMEDIACONTROLDIGIT**. Each time a digit listed in the digit list is detected, the specified media control action is carried out on the call's media stream.

Valid digits for pulse mode are '0' through '9'. Valid digits for DTMF mode are '0' through '9', 'A', 'B', 'C', 'D', '*', '#'.

dwDigitNumEntries

Specifies the number of entries in the *lpDigitList*. Note that this parameter is not validated by TAPI.DLL when this function is called.

lpMediaList

Specifies a far pointer to an array with entries of type **LINEMEDIACONTROLMEDIA**. The array has *dwMediaNumEntries* entries. Each entry contains a media mode to be monitored, media-type specific information (such as duration), and a media control field. If a media mode in the list is detected, the corresponding media control action is performed on the call's media stream.

dwMediaNumEntries

Specifies the number of entries in *lpMediaList*. Note that this parameter is not validated by TAPI.DLL when this function is called.

lpToneList

Specifies a far pointer to an array with entries of type **LINEMEDIACONTROLTONE**. The array has *dwToneNumEntries* entries. Each entry contains a description of a tone to be monitored, duration of the tone, and a media control field. If a tone in the list is detected, the corresponding media control action is performed on the call's media stream.

dwToneNumEntries

Specifies the number of entries in *lpToneList*. Note that this parameter is not validated by TAPI.DLL when this function is called.

lpCallStateList

Specifies a far pointer to an array with entries are of type **LINEMEDIACONTROLCALLSTATE**. The array has *dwCallStateNumEntries* entries. Each entry contains a call state and a media control action.

dwCallStateNumEntries

Specifies the number of entries in *lpCallStateList*. Note that this parameter is not validated by TAPI.DLL when this function is called.

This function returns zero (success) when media control has been correctly initiated; not when any media control has taken effect. Media control in progress is changed or is canceled when this function is called again with either different parameters or NULLs.

Only a single media control request can be outstanding on a call at one time. A request replaces any that may be outstanding.

Depending on the service provider and other activities that compete for such resources, the amount of simultaneous detections that can be made may vary over time. If service provider resources are overcommitted, it returns LINEERR_RESOURCEUNAVAIL.

Whether or not media control is supported by the service provider is a device capability indicated in **LINEDEVCAPS**.

Each time **TSPI_lineSetMediaControl** is called, the new request overrides any media control currently in effect. If one or more of the parameters *lpDigitList*, *lpMediaList*, *lpToneList*, and *lpCallStateList* are NULL, the corresponding digit, media mode, tone, or call state-triggered media control is disabled. To modify just a portion of the media control parameters while leaving the remaining settings in effect, the application should invoke **TSPI_lineSetMediaControl** supplying the previous parameters for those potions that must remain in effect, and new parameters for those parts that are to be modified.

TSPI_lineSetMediaMode

[New - Windows 95]

```
LONG TSPI_lineSetMediaMode(HDRVCALL hdCall, DWORD dwMediaMode)
```

Changes the call's media as stored in the call's **LINECALLINFO** structure.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALMEDIAMODE	LINEERR_OPERATIONFAILED
LINEERR_NOMEM	LINEERR_RESOURCEUNAVAIL

hdCall

Specifies the handle to the call undergoing a change in media mode. The call state of *hdCall* can be any state.

dwMediaMode

Specifies the new media mode(s) for the call. As long as the LINEMEDIAMODE_UNKNOWN media mode flag is set, multiple other media mode flags may be set as well. This is used to identify a call's media mode as not fully determined, but narrowed down to one of just a small set of specified media modes. If the LINEMEDIAMODE_UNKNOWN flag is not set, only a single media mode can be specified. LINEMEDIAMODE_ constant values are:

LINEMEDIAMODE_UNKNOWN
The anticipated media type is not fully classified.

LINEMEDIAMODE_INTERACTIVEVOICE
The anticipated media type is interactive voice media mode, namely live conversations.

LINEMEDIAMODE_AUTOMATEDVOICE
TAPI.DLL wants to handle calls in which there is voice data, but the voice is handled by an automated application. Most service providers will not be able to distinguish between automated and interactive voice. In such a case any request for voice or detection of voice should be treated as interactive voice.

LINEMEDIAMODE_DATAMODEM
The anticipated media type is data modem media mode.

LINEMEDIAMODE_G3FAX
The anticipated media type is group 3 fax media mode.

LINEMEDIAMODE_TDD
The anticipated media type is TDD (Telephony Devices for the Deaf) media mode.

LINEMEDIAMODE_G4FAX
The anticipated media type is group 4 fax media mode.

LINEMEDIAMODE_DIGITALDATA
The anticipated media type is digital data calls.

LINEMEDIAMODE_TELETEX
The anticipated media type is teletex media mode.

LINEMEDIAMODE_VIDEOTEX
The anticipated media type is videotex media mode.

LINEMEDIAMODE_TELEX
The anticipated media type is telex media mode.

LINEMEDIAMODE_MIXED
The anticipated media type is ISDN mixed media mode.

LINEMEDIAMODE_ADSI
The anticipated media type is ADSI (Analog Display Services Interface) media mode.

Other than changing the call's media as stored in the call's **LINECALLINFO** structure, this procedure is simply "advisory" in the sense that it indicates an expected media change that is about to occur, rather than forcing a specific change to the call. Typical usage is to set a call's media mode to a specific known media mode, or to exclude possible media modes as long as the call's media mode is not fully known (the UNKNOWN media mode flag is set).

TAPI.DLL makes the following guarantees regarding the passed media mode: (1) there is at least one bit set, (2)

there are no reserved bits set, and (3) if more than one bit is set, "Unknown" is also set. It is the responsibility of the service provider to do any further validity checks on the media modes, such as checking whether any regular or extended media modes are indeed supported by the service provider.

TSPI_lineSetStatusMessages

[New - Windows 95]

```
LONG TSPI_lineSetStatusMessages(HDRVLINE hdLine, DWORD dwLineStates)
```

Enables TAPI.DLL to specify which notification messages the service provider should generate for events related to status changes for the specified line or any of its addresses.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

  LINEERR_INVALADDRESSSTATE LINEERR_OPERATIONUNAVAIL
  LINEERR_INVALLINEHANDLE LINEERR_OPERATIONFAILED
  LINEERR_INVALLINESTATE LINEERR_RESOURCEUNAVAIL
  LINEERR_NOMEM

hdLine

Specifies the handle to the line device for which the new filter is to be set.

dwLineStates

Specifies a bit array that identifies for which line device status changes a message is to be sent to TAPI.DLL. This parameter uses the following LINEDEVSTATE_. constants:

LINEDEVSTATE_OTHER
Device-status items other than those listed below have changed.

LINEDEVSTATE_RINGING
The switch tells the line to alert the user.

LINEDEVSTATE_CONNECTED
The line was previously disconnected and is now connected to the service provider.

LINEDEVSTATE_DISCONNECTED
This line was previously connected and is now disconnected from the service provider.

LINEDEVSTATE_MSGWAITON
The “message waiting” indicator is turned on.

LINEDEVSTATE_MSGWAITOFF
The “message waiting” indicator is turned off.

LINEDEVSTATE_INSERVICE
The line is connected to the service provider. This happens when the service provider is first activated, or when the line wire is physically plugged in and in service at the switch while the service provider is active.

LINEDEVSTATE_OUTOFSERVICE
The line is out of service at the switch or physically disconnected. The API cannot be used to operate on the line device.

LINEDEVSTATE_MAINTENANCE
Maintenance is being performed on the line at the switch. The service provider cannot be used to operate on the line device.

LINEDEVSTATE_NUMCALLS
The number of calls on the line device has changed.

LINEDEVSTATE_NUMCOMPLETIONS
The number of outstanding call completions on the line device has changed.

LINEDEVSTATE_TERMINALS
The terminal settings have changed.

LINEDEVSTATE_ROAMMODE
The roaming mode of the line device has changed.

LINEDEVSTATE_BATTERY
The battery level has changed significantly (cellular).

LINEDEVSTATE_SIGNAL
The signal level has changed significantly (cellular).

LINEDEVSTATE_DEVSPECIFIC
The line’s device-specific information has changed.

LINEDEVSTATE_REINIT

Items have changed in the configuration of line devices. To become aware of these changes (as with the appearance of new line devices) TAPI.DLL should notify all its client applications to reinitialize their use of the API. A service provider should initiate such a sequence only in extreme cases that cannot be handled through the normal configuration-change notification mechanism described in the “Overview” chapter of the TSPI documentation. The LINEDEVSTATE_REINIT message is never masked. REINIT messages should always be sent when appropriate, regardless of the setting of this flag. The setting of the REINIT flag is simply ignored and generates no error.

LINEDEVSTATE_LOCK
The locked status of the line device has changed.

DWORD *dwAddressStates*

Specifies a bit array that identifies for which address status changes a message is to be sent to TAPI.DLL. This parameter uses the following LINEADDRESSSTATE_ constants:

LINEADDRESSSTATE_OTHER
Address-status items other than those listed below have changed.

LINEADDRESSSTATE_DEVSPECIFIC
The device-specific item of the address status has changed.

LINEADDRESSSTATE_INUSEZERO
The address has changed to idle (it is now in use by zero stations).

LINEADDRESSSTATE_INUSEONE
The address has changed from being idle or from being in use by many bridged stations to being in use by just one station.

LINEADDRESSSTATE_INUSEMANY
The monitored or bridged address has changed to being in use by one station to being used by more than one station.

LINEADDRESSSTATE_NUMCALLS
The number of calls on the address has changed, as a result of an event such as a new inbound call, an outbound call on the address, or a call changing its hold status.

LINEADDRESSSTATE_FORWARD
The forwarding status of the address has changed including the number of rings for determining a no answer condition. TAPI.DLL should check the address status to determine details about the address’s current forwarding status.

LINEADDRESSSTATE_TERMINALS
The terminal settings for the address have changed.

The service provider returns LINEERR_INVALLINESTATE if the *dwLineStates* parameter contains one or more bits that are not LINEDEVSTATE_ constants.

Windows Telephony defines a number of messages that notify applications about events occurring on lines and addresses. The sets of all change messages in which all applications are interested may be much smaller than the set of possible messages. This procedure allows TAPI.DLL to tell the service provider the reduced set of messages that should be delivered. The service provider should deliver all of the messages it supports, within the specified set. It is permitted to deliver more (they will be filtered out by TAPI.DLL), but is discouraged from doing so for performance reasons. If TAPI.DLL requests delivery of a particular message type that is not produced by the provider, the provider should nevertheless accept the request but simply not produce the message. By default, address and line status reporting is initially disabled for a line.

This function differs from the corresponding TAPI function as follows: (1) The set of messages requested is the union of all sets requested by applications at the TAPI level. (2) The message set is neither reduced nor augmented by ownership (since there is no concept of ownership at the TSPI level) (3) The set is “advisory” in the sense that the service provider is required to forward at least the indicated set of messages but is permitted to forward a larger set.

Device state changes regarding Open and Close are not reported, since at the TSPI level there is only one outstanding Open at a time.

TSPI_lineSetTerminal

[New - Windows 95]

```
LONG TSPI_lineSetTerminal(DRV_REQUESTID dwRequestID, HDRVLINE hdLine,
    DWORD dwAddressID, HDRVCALL hdCall, DWORD dwSelect,
    DWORD dwTerminalModes, DWORD dwTerminalID, DWORD bEnable)
```

Enables TAPI.DLL to specify to which terminal information related to the specified line, address, or call is to be routed. This operation can be used while calls are in progress on the line, to allow events to be routed to different devices as required.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALLINEHANDLE	LINEERR_INVALTERMINALID
LINEERR_INVALADDRESSID	LINEERR_RESOURCEUNAVAIL
LINEERR_INVALCALLHANDLE	LINEERR_NOMEM
LINEERR_INVALCALLSELECT	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALTERMINALMODE	LINEERR_OPERATIONFAILED

dwRequestID

Specifies the identifier of the asynchronous request.

hdLine

Specifies the handle to a line.

dwAddressID

Specifies an address on the given open line device. This parameter is not validated by TAPI.DLL when this function is called.

hdCall

Specifies the handle to a call. The call state can be any state (if *dwSelect* is LINECALLSELECT_CALL).

dwSelect

Specifies whether the terminal setting is requested for the line, the address, or just the specified call. If line or address is specified, events either apply to the line or address itself or serve as a default initial setting for all new calls on the line or address. This parameter uses the following LINECALLSELECT_ constants:

LINECALLSELECT_LINE

Selects the specified line device. The *hdLine* parameter must be a valid line handle; *hdCall* and *dwAddressID* must be ignored.

LINECALLSELECT_ADDRESS

Selects the specified address on the line. Both *hdLine* and *dwAddressID* must be valid; *hdCall* must be ignored. The service provider must validate *dwAddressID*; TAPI.DLL validates *hdLine*.

LINECALLSELECT_CALL

Selects the specified call. *hdCall* must be valid; *hdLine* and *dwAddressID* must both be ignored.

dwTerminalModes

Specifies the class(es) of low level events to be routed to the given terminal.

LINETERMMODE_BUTTONS

The button presses from the terminal to the line.

LINETERMMODE_LAMPS

The lamp lighting events from the line to the terminal.

LINETERMMODE_DISPLAY

The display events from the line to the terminal.

LINETERMMODE_RINGER

The ring requests from the line to the terminal.

LINETERMMODE_HOOKSWITCH

The hookswitch events between the terminal and the line.

LINETERMMODE_MEDIATOLINE

This is the unidirectional media stream from the terminal to the line associated with a call on the line. This

value is only allowed when routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIAFROMLINE
This is the unidirectional media stream from the line to the terminal associated with a call on the line. This value is only allowed when routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIABIDIRECT
This is the bidirectional media stream associated with a call on the line and the terminal. This is the only value allowed when routing of both unidirectional channels of a call's media stream cannot be controlled independently. Note that the media modes (1) MEDIABIDIRECT and (2) MEDIATOLINE and MEDIAFROMLINE are exclusive, the enforcement of which is the responsibility of the service provider.

dwTerminalID

Specifies the device ID of the terminal device where the given events are to be routed. Terminal IDs are small integers in the range of 0 to **dwNumTerminals** minus one, where **dwNumTerminals** and the terminal modes each terminal is capable of handling are indicated by the service provider in **LINEDEVCAPS**. Note that these terminal IDs have no relation to other device IDs and are defined by the service provider through device capabilities. This parameter is not validated by TAPI.DLL when this function is called.

bEnable

If TRUE, *dwTerminalID* is valid and the specified event classes are routed to or from that terminal. If FALSE, these events are not routed to or from the *dwTerminalID*. Note that this parameter is not validated by TAPI.DLL when this function is called.

The service provider returns LINEERR_RESOURCEUNAVAIL if the operation cannot be completed because of resource overcommitment or if too many terminals are set, due either to hardware limitations or to service provider/device driver limitations.

TAPI.DLL can use this operation to route certain classes of low level line events to the specified terminal device, or to suppress the routing of these events altogether. For example, voice may be routed to a separate audio I/O device (headset), lamps and display events may be routed to the local phone device, and button events and ringer events may be suppressed altogether.

It is the responsibility of the service provider to determine whether the combinations of *dwSelect* and *dwTerminalModes* are legal.

This operation can be called any time, even when a call is active on the given line device. This, for example, allows a user to switch from using the local phone set to another audio I/O device.

This function may be called multiple times to route the same events to multiple terminals simultaneously. To reroute events to a different terminal, TAPI recommends that the application first disable routing to the existing terminal and next route the events to the new terminal. However, the service provider should make its best effort to accomodate the application's requests in any sequence.

Terminal ID assignments are made by the service provider, and **LINEDEVCAPS** indicates which terminal IDs the service provider has available. Service providers that don't support this type of event routing indicate that they have no terminal devices (**dwNumTerminals** in **LINEDEVCAPS** is set to zero).

LineSetTerminal on a line or address affects all existing calls on that line or address, but does not affect calls on other addresses. It also sets the default for future calls on that line or address. A line or address that has multiple connected calls active at any one time may have different routing in effect for each call.

Disabling the routing of low-level events to a terminal when these events are not currently routed to or from that terminal is not required to generate an error so long as after the function succeeds, the specified events are not routed to or from that terminal.

TSPI_lineSetupConference

[New - Windows 95]

```
LONG TSPI_lineSetupConference(DRV_REQUESTID dwRequestID,
    HDRVCALL hdCall, HDRVLINE hdLine, HTAPICALL htConfCall,
    LPHDRVCALL lphdConfCall, HTAPICALL htConsultCall,
    LPHDRVCALL lphdConsultCall, DWORD dwNumParties,
    LPLINECALLPARAMS const lpCallParams)
```

Sets up a conference call for the addition of the third party.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_INVALADDRESSMODE
LINEERR_INVALLINEHANDLE	LINEERR_INVALBEARERMODE
LINEERR_INVALCALLSTATE	LINEERR_INVALCALLPARAMS
LINEERR_CALLUNAVAIL	LINEERR_INVALLINESTATE
LINEERR_CONFERENCEFULL	LINEERR_INVALMEDIAMODE
LINEERR_NOMEM	LINEERR_INVALRATE
LINEERR_OPERATIONUNAVAIL	LINEERR_INUSE
LINEERR_OPERATIONFAILED	LINEERR_RATEUNAVAIL
LINEERR_RESOURCEUNAVAIL	LINEERR_USERUSERINFOTOOBIG
LINEERR_BEARERMODEUNAVAIL	

dwRequestID

Specifies the identifier of the asynchronous request.

hdCall

Specifies the handle to the initial call that identifies the first party of a conference call. In some environments, a call must exist in order to start a conference call. In other telephony environments, no call initially exists and *hdCall* is left NULL. The call state of *hdCall* can be *connected*.

hdLine

Specifies the handle to the line device on which to originate the conference call if *hdCall* is NULL. The *hdLine* parameter is ignored if *hdCall* is non-NULL. The service provider reports which model it supports through the **setupConfNull** flag of the **LINEADDRESSCAPS** data structure.

htConfCall

Specifies TAPI.DLL's handle to the new conference call. The service provider must save this and use it in all subsequent calls to the **LINEEVENT** procedure reporting events on the new call.

lphdConfCall

Specifies a far pointer to an HDRVCALL representing the service provider's identifier for the newly created conference call. The service provider must fill this location with its handle for the new call before this procedure returns. This handle is ignored by TAPI.DLL if the function results in an error. The call state of *hdConfCall* is not applicable.

htConsultCall

Specifies TAPI.DLL's handle to the consultation call. When setting up a call for the addition of a new party, a new temporary call (consultation call) is automatically allocated. The service provider must save the *htConsultCall* and use it in all subsequent calls to the **LINEEVENT** procedure reporting events on the new consultation call.

lphdConsultCall

Specifies a far pointer to an HDRVCALL representing the service provider's identifier for a call. When setting up a call for the addition of a new party, a new temporary call (consultation call) is automatically allocated. The service provider must fill this location with its handle for the new consultation call before this procedure returns. This handle is ignored by TAPI.DLL if the function results in an error. The call state of *hdConsultCall* is not applicable.

dwNumParties

Specifies the expected number of parties in the conference call. The service provider is free to do with this number as it pleases. For example, the service provider can ignore it, or use it as a hint to allocate the right size conference bridge inside the switch. Note that this parameter is not validated by TAPI.DLL when this function is called.

lpCallParams

Specifies a far pointer to call parameters to be used when establishing the consultation call. This parameter will be set to NULL if no special call setup parameters are desired and the service provider uses default parameters.

The service provider returns LINEERR_INVALLINEHANDLE if the specified line handle for the line containing the conference call is invalid. This error may also indicate that the telephony environment requires an initial line to set up a conference but a non-NULL call handle was supplied instead.

The service provider returns LINEERR_INVALCALLHANDLE if the telephony environment requires an initial call to set up a conference but a NULL call handle was supplied.

TSPI_lineSetupConference provides two ways to establish a new conference call, depending on whether a normal two-party call is required to pre-exist or not. When setting up a conference call from an existing two-party call, the *hdCall* parameter is a valid call handle that is initially added to the conference call by the **TSPI_lineSetupConference** request and *hdLine* is ignored. On switches where conference call setup does not start with an existing call, *hdCall* must be NULL and *hdLine* must be specified to identify the line device on which to initiate the conference call. In either case, a consultation call is allocated for connecting to the party that is to be added to the call. TAPI.DLL can use **TSPI_lineDial** to dial the address of the other party.

The conference call will typically transition into the *onHoldPendingConference* state, the consultation call *dialtone* state and the initial call (if one) into the *conferenced* state.

A conference call can also be set up using a **TSPI_lineCompleteTransfer** that is resolved into a three-way conference.

TAPI.DLL may be able to toggle between the consultation call and the conference call using **TSPI_lineSwapHold**.

A consultation call can be canceled by invoking **TSPI_lineDrop** on it. When dropping a consultation call, the existing conference call will typically transition back to the *connected* state. TAPI.DLL and its client applications should observe the **LINE_CALLSTATE** messages to determine exactly what happens to the calls. For example, if the conference call reverts back to a regular two party call, the conference call will become *idle* and the original participant call may revert to *connected*.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

This function differs from the corresponding TAPI function in that it follows the TSPI model for beginning the lifetime of a call. TAPI.DLL and the service provider exchange opaque handles representing the call with one another. In addition, the service provider is permitted to do callbacks for the new call before it returns from this procedure. In any case, the service provider must also treat the handle it returned as “not yet valid” until after the matching **ASYNC_COMPLETION** message reports success. In other words, it must not issue any **LINEEVENT** messages for the new call or include it in call counts in messages or status data structures for the line.

TSPI_lineSetupTransfer

[New - Windows 95]

```
LONG TSPI_lineSetupTransfer(DRV_REQUESTID dwRequestID, HDRVCALL hdCall,
    HTAPICALL htConsultCall, LPHDRVCALL lphdConsultCall,
    LPLINECALLPARAMS const lpCallParams)
```

Initiates a transfer of the call specified by *hdCall*. It establishes a consultation call, *lphdConsultCall*, on which the party can be dialed that can become the destination of the transfer.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_INVALBEARERMODE
LINEERR_INVALCALLSTATE	LINEERR_INVALRATE
LINEERR_CALLUNAVAIL	LINEERR_INVALCALLPARAMS
LINEERR_NOMEM	LINEERR_INVALLINESTATE
LINEERR_OPERATIONUNAVAIL	LINEERR_INVALMEDIAMODE
LINEERR_OPERATIONFAILED	LINEERR_INUSE
LINEERR_RESOURCEUNAVAIL	LINEERR_NOMEM
LINEERR_BEARERMODEUNAVAIL	LINEERR_RATEUNAVAIL
LINEERR_INVALADDRESSMODE	LINEERR_USERUSERINFOTOOBIG

dwRequestID

Specifies the identifier of the asynchronous request.

hdCall

Specifies the handle to the call to be transferred. The call state of *hdCall* can be *connected*.

htConsultCall

Specifies TAPI.DLL's handle to the new, temporary consultation call. The service provider must save this and use it in all subsequent calls to the **LINEEVENT** procedure reporting events on the new consultation call.

lphdConsultCall

Specifies a far pointer to an HDRVCALL representing the service provider's identifier for the new consultation call. The service provider must fill this location with its handle for the new consultation call before this procedure returns. This handle is ignored by TAPI.DLL if the function results in an error. The call state of *hdConsultCall* is not applicable.

When setting a call up for transfer, another call (a consultation call) is automatically allocated to enable the application (through TAPI.DLL) to dial the address (using **TSPI_lineDial**) of the party to where the call is to be transferred. The originating party can carry on a conversation over this consultation call prior to completing the transfer.

This transfer procedure may not be valid for some line devices. Instead of calling this procedure, TAPI.DLL may need to unhold an existing held call (using **TSPI_lineUnhold**) to identify the destination of the transfer. On switches that support cross-address call transfer, the consultation call may exist on a different address than the call to be transferred. It may also be necessary to set up the consultation call as an entirely new call using **TSPI_lineMakeCall**, to the destination of the transfer.

The **transferHeld** and **transferMake** flags in the **LINEADDRESSCAPS** data structure report what model the service provider uses.

lpCallParams

Specifies a far pointer to call parameters to be used when establishing the consultation call. This parameter may be set to NULL if no special call setup parameters are desired (the service provider uses defaults).

The service provider returns LINEERR_INVALCALLSTATE if the call to be transferred is not in a valid state.

This operation sets up the transfer of the call specified by *hdCall*. The setup phase of a transfer establishes a consultation call to send the address of the destination (the party to be transferred to) to the switch, while the call to be transferred is kept on hold. This new call is referred to as a *consultation call* (*hdConsultCall*) and can be

manipulated (for example, dropped) independently of the original call.

When the consultation call has reached the *dialtone* call state, TAPI.DLL may continue transferring the call either by dialing the destination address and tracking its progress, or by unholding an existing call. The transfer of the original call to the selected destination is completed using **TSPI_lineCompleteTransfer**.

While the consultation call exists, the original call will typically transition to the *onholdPendingTransfer* state.

In telephony environments that follow the **transferHeld** or **transferMake** transfer models, this procedure returns LINEERR_OPERATIONFAILED and does not allocate a consultation call handle.

A consultation call can be canceled by invoking **TSPI_lineDrop** on it. After dropping a consultation call, the original call will typically transition back to the *connected* state.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

This function differs from the corresponding TAPI function in that it follows the TSPI model for beginning the lifetime of a call. TAPI.DLL and the service provider exchange opaque handles representing the call with one another. In addition, the service provider is permitted to do callbacks for the new call before it returns from this procedure. In any case, the service provider must also treat the handle it returned as “not yet valid” until after the matching **ASYNC_COMPLETION** message reports success. In other words, it must not issue any **LINEEVENT** messages for the new call or include it in call counts in messages or status data structures for the line.

TSPI_lineSwapHold

[New - Windows 95]

```
LONG TSPI_lineSwapHold(DRV_REQUESTID dwRequestID, HDRVCALL hdActiveCall,
    HDRVCALL hdHeldCall)
```

Swaps the specified active call with the specified call on consultation hold.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALCALLSTATE	LINEERR_OPERATIONFAILED
LINEERR_NOMEM	LINEERR_RESOURCEUNAVAIL

dwRequestID

Specifies the identifier of the asynchronous request.

hdActiveCall

Specifies the handle to the call to be swapped with the call on consultation hold. The call state of *hdActiveCall* can be *connected*.

hdHeldCall

Specifies the handle to the consultation call. The call state of *hdHeldCall* can be *onHoldPendingTransfer*, *onHoldPendingConference*, *onHold*.

The service provider must send **LINECALLSTATE** messages for the call transitions.

TSPI_lineUncompleteCall

[New - Windows 95]

```
LONG TSPI_lineUncompleteCall(DRV_REQUESTID dwRequestID, HDRVLINE hdLine,
    DWORD dwCompletionID)
```

Is used to cancel the specified call competion request on the specified line.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALLINEHANDLE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALCOMPLETIONID	LINEERR_OPERATIONFAILED
LINEERR_NOMEM	LINEERR_RESOURCEUNAVAIL

dwRequestID

Specifies the identifier of the asynchronous request.

hdLine

Specifies the handle to the line on which a call completion is to be canceled.

dwCompletionID

Specifies the completion ID for the request that is to be canceled. This parameter is not validated by TAPI.DLL when this function is called.

TSPI_lineUnhold

[New - Windows 95]

```
LONG TSPI_lineUnhold(DRV_REQUESTID dwRequestID, HDRVCALL hdCall)
```

Retrieves the specified held call.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALCALLHANDLE	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALCALLSTATE	LINEERR_OPERATIONFAILED
LINEERR_NOMEM	LINEERR_RESOURCEUNAVAIL

dwRequestID

Specifies the identifier of the asynchronous request.

hdCall

Specifies the handle to the call to be retrieved. The call state of *hdCall* can be *onHold*.

The service provider returns LINEERR_INVALCALLSTATE if the call is not currently on hold.

This operation works only for calls on hard hold (calls placed on hold using **TSPI_lineHold**). The service provider should check that the call is currently in the *onHold* state, change the state to *connected*, and send a **LINECALLSTATE** message for the new call state.

TSPI_lineUnpark

[New - Windows 95]

```
LONG TSPI_lineUnpark(DRV_REQUESTID dwRequestID, HDRVLINE hdLine,
    DWORD dwAddressID, HTAPICALL htCall, LPHDRVCALL lphdCall,
    LPCSTR lpszDestAddress)
```

Retrieves the call parked at the specified address and returns a call handle for it.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

LINEERR_INVALLINEHANDLE	LINEERR_NOMEM
LINEERR_INVALPOINTER	LINEERR_OPERATIONUNAVAIL
LINEERR_INVALADDRESSID	LINEERR_OPERATIONFAILED
LINEERR_INVALADDRESS	LINEERR_RESOURCEUNAVAIL

dwRequestID

Specifies the identifier of the asynchronous request.

hdLine

Specifies the handle to the line on which a call is to be unparked.

dwAddressID

Specifies the address on *hdLine* at which the unpark is to be originated. This parameter is not validated by TAPI.DLL when this function is called.

htCall

Specifies TAPI.DLL's handle to the new unparked call. The service provider must save this and use it in all subsequent calls to the **LINEEVENT** procedure reporting events on the call.

lphdCall

Specifies a far pointer to an HDRVCALL representing the service provider's identifier for the new unparked call. The service provider must fill this location with its handle for the call before this procedure returns. This handle is invalid if the function results in an error.

lpszDestAddress

Specifies a far pointer to a NULL terminated ASCII string that contains the address where the call is parked. The address is in dialable address format.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

This function differs from the corresponding TAPI function in that it follows the TSPI model for beginning the lifetime of a call. TAPI.DLL and the service provider exchange opaque handles representing the call with one another. In addition, the service provider is permitted to do callbacks for the new call before it returns from this procedure. In any case, the service provider must also treat the handle it returned as "not yet valid" until after the matching **ASYNC_COMPLETION** message reports success. In other words, it must not issue any **LINEEVENT** messages for the new call or include it in call counts in messages or status data structures for the line.

The call handle created by this function is a new, distinct, call handle even if an original call handle for the call is still in existence (it has not been destroyed by **TSPI_lineCloseCall**).

Line Messages

Line Messages

About Line Messages

This chapter contains a list of the messages in the Telephony Service Provider Interface (TSPI). Messages are used to notify TAPI.DLL of the occurrence of asynchronous events that spontaneously occur within the service provider. The service provider passes these events to TAPI.DLL by calling a **LINEEVENT** or **PHONEEVENT** callback function depending on whether the service provider is reporting an event on a line, call or phone device. The **LINEEVENT** procedure for reporting events occurring on a line or call is supplied to the service provider at the time the line is opened with the **TSPI_lineOpen** function. The **PHONEEVENT** procedure for reporting events occurring on a phone is supplied with the **TSPI_phoneOpen** function.

These spontaneous events are unsolicited by TAPI.DLL in the sense that they are not a direct response to any request. These contrast with events reporting completion of requests made by TAPI.DLL. Such completion events are reported through the **ASYNC_COMPLETION** callback function as described in the “Functions” chapter.

The parameter profiles for the spontaneous event procedures include parameters that identify the relevant object for which the event is being reported (phone, line, or call). The identification is in the form of an opaque handle whose exact interpretation is not published by the TSPI. TAPI.DLL internally determines the relationship between these opaque handles and whatever data structures it used to represent the devices.

The parameter profile for spontaneous event procedures also includes a message parameter identifying the type of the message. Each message type has a corresponding definition that determines what handles are included, along with what other parameters and their meanings. There is a very strong correspondence between the messages appearing at the TSPI level and those that appear at the TAPI level. The general rules of correspondence are as follows:

- The set of messages is nearly identical. Where messages correspond, the same message name and value is used at the TSPI level.
- Handles appearing at the TSPI level are the opaque types defined by the TSPI specification. These types (and their interpretation) differ from those at the TAPI level, although they refer to the same class of device. For example, where a TAPI message would include an HLINE handle, the corresponding TSPI message would typically include an HTAPILINE handle.
- There is no *dwCallbackInstance* data passed to the callback.
- *dwParam1*, *dwParam2*, and *dwParam3* are usually identical to the corresponding parameters for the TAPI message.
- line-oriented and call-oriented messages are passed to a different callback procedure than phone-oriented messages.

For each message, this chapter lists the following items:

- The purpose of the message
- The callback procedure to which this message is passed
- A description of the message parameters
- Optional comments about using the message
- Optional references to other functions, messages, and data structures
- Optional comments comparing this message to the TAPI interface

Certain messages are used to notify TAPI.DLL about a change in an object’s status. These messages provide TAPI.DLL’s opaque object handle and an indication of which status item has changed. TAPI.DLL can subsequently call an appropriate “get status” function of the object to obtain the object’s full status.

When an event occurs, a message may or may not be sent to TAPI.DLL. For some event types, such as status changes, TAPI.DLL specifies a set of status changes in which it is interested. The service provider is advised to limit the status-change messages events it reports to those included in this set. The service provider is not required to adhere to this limit. In other words, it may report more changes than are strictly necessary. However, it should try to observe the limit for performance reasons.

The LINE_REPLY message is not used at the TSPI level. Completion of an asynchronous request is reported using the **ASYNC_COMPLETION** callback.

Reference

LINE_ADDRESSSTATE

[New - Windows 95]

```
htLine = (HTAPILINE) hLineDevice;
htCall = (HTAPICALL) 0;
dwMsg = (DWORD) LINE_ADDRESSSTATE;
dwParam1 = (DWORD) idAddress;
dwParam2 = (DWORD) AddressState;
dwParam3 = (DWORD) 0;
```

Sent to the **LINEEVENT** callback when the status of an address changes on a line that is currently open by TAPI.DLL. TAPI.DLL can invoke **TSPI_lineGetAddressStatus** to determine the current status of the address.

htLine

Specifies TAPI.DLL's opaque object handle to the line device.

htCall

Unused

dwMsg

The value **LINE_ADDRESSSTATE**

dwParam1

Specifies the address ID of the address that changed status.

dwParam2

Specifies the address state that changed. This parameter uses the following LINEADDRESSSTATE_constants:

LINEADDRESSSTATE_OTHER
: Address-status items other than those listed below have changed. TAPI.DLL can check the current address status to determine which items have changed.

LINEADDRESSSTATE_DEVSPECIFIC
: The device-specific item of the address status has changed.

LINEADDRESSSTATE_INUSEZERO
: The address has changed to idle (it is now in use by zero stations).

LINEADDRESSSTATE_INUSEONE
: The address has changed from being idle or from being in use by many bridged stations to being in use by just one station.

LINEADDRESSSTATE_INUSEMANY
: The monitored or bridged address has changed to being in use by one station to being used by more than one station.

LINEADDRESSSTATE_NUMCALLS
: The number of calls on the address has changed. This is the result of an event such as a new inbound call, an outbound call on the address, or a call changing its hold status.

LINEADDRESSSTATE_FORWARD
: The forwarding status of the address has changed including the number of rings for determining a no answer condition. The application should check the address status to determine details about the address's current forwarding status.

LINEADDRESSSTATE_TERMINALS
: The terminal settings for the address have changed.

dwParam3

Unused.

This message is sent whenever the line is open by TAPI.DLL and an event occurs in which TAPI.DLL has expressed an interest. TAPI.DLL uses the **TSPI_lineSetStatusMessages** function to specify the set of status-change events in which it is interested. By default, address status reporting is disabled.

LINE_CALLDEVSPECIFIC

[New - Windows 95]

```
htLine = (HTAPILINE) hLineDevice;
htCall = (HTAPICALL) hCallDevice;
dwMsg = (DWORD) LINE_CALLDEVSPECIFIC;
dwParam1 = (DWORD) DeviceData1;
dwParam2 = (DWORD) DeviceData2;
dwParam3 = (DWORD) DeviceData3;
```

Sent to the **LINEEEVENT** callback to notify TAPI.DLL about device-specific events occurring on a call. The meaning of the message and the interpretation of the *dwParam1* through *dwParam3* parameters is device specific.

htLine
Specifies TAPI.DLL's opaque object handle to the line device.

htCall
Specifies TAPI.DLL's opaque object handle to the call device.

dwMsg
The value **LINE_CALLDEVSPECIFIC**

dwParam1
Device specific.

dwParam2
Device specific.

dwParam3
Device specific.

This message is used by a service provider in conjunction with the **TSPI_lineDevSpecific** function. Its meaning is device specific.

TAPI.DLL sends the **LINE_DEVSPECIFIC** message to applications in response to receiving this message from a service provider. The *htLine* and *htCall* are translated to the appropriate *hCall* as the *hDevice* parameter at the TAPI level. The *dwParam1, dwParam2,* and *dwParam3* paramters are passed through unmodified.

There is no directly corresponding message at the TAPI level, although this message is very similar to **LINE_DEVSPECIFIC**. At the TSPI level, the device-specific messages are split between those reporting events on lines and addresses, and those reporting events on calls.

LINE_CALLDEVSPECIFICFEATURE

[New - Windows 95]

```
htLine = (HTAPILINE) hLineDevice;
htCall = (HTAPICALL) hCallDevice;
dwMsg = (DWORD) LINE_CALLDEVSPECIFICFEATURE;
dwParam1 = (DWORD) DeviceData1;
dwParam2 = (DWORD) DeviceData2;
dwParam3 = (DWORD) DeviceData3;
```

Sent to the **LINEEVENT** callback to notify TAPI.DLL about device-specific events occurring a line or address. The meaning of the message and the interpretation of the *dwParam1* through *dwParam3* parameters is device specific.

htLine

Specifies TAPI.DLL's opaque object handle to the line device.

htCall

Specifies TAPI.DLL's opaque object handle to the call device.

dwMsg

The value **LINE_CALLDEVSPECIFICFEATURE**

dwParam1

Device specific.

dwParam2

Device specific.

dwParam3

Device specific.

This message is used by a service provider in conjunction with the **TSPI_lineDevSpecificFeature** function. Its meaning is device specific

TAPI.DLL sends the **LINE_DEVSPECIFICFEATURE** message to applications in response to receiving this message from a service provider. The *htLine* and *htCall* are translated to the appropriate *hCall* as the *hDevice* parameter at the TAPI level. The *dwParam1, dwParam2,* and *dwParam3* paramters are passed through unmodified.

There is no directly corresponding message at the TAPI level, although this message is very similar to **LINE_DEVSPECIFICFEATURE**. At the TSPI level, the device-specific feature messages are split between those reporting events on lines and addresses, and those reporting events on calls.

LINE_CALLINFO

[New - Windows 95]

```
htLine = (HTAPILINE) hLineDevice;
htCall = (HTAPICALL) hCallDevice;
dwMsg = (DWORD) LINE_CALLINFO;
dwParam1 = (DWORD) CallInfoState;
dwParam2 = (DWORD) 0;
dwParam3 = (DWORD) 0;
```

Sent to the **LINEEVENT** callback when the call information about the specified call has changed. TAPI.DLL can invoke **lineGetCallInfo** to determine the current call information.

htLine

Specifies TAPI.DLL's opaque object handle to the line device.

htCall

Specifies TAPI.DLL's opaque object handle to the call device.

dwMsg

The value LINE_CALLINFO

dwParam1

Specifies the call information item that has changed. This parameter uses the following LINECALLINFOSTATE_ constants:

LINECALLINFOSTATE_OTHER
Information items other than those listed below have changed.

LINECALLINFOSTATE_DEVSPECIFIC
The device-specific field of the call-information record.

LINECALLINFOSTATE_BEARERMODE
The bearer mode field of the call-information record.

LINECALLINFOSTATE_RATE
The rate field of the call-information record.

LINECALLINFOSTATE_MEDIAMODE
The media mode field of the call-information record.

LINECALLINFOSTATE_APPSPECIFIC
The application-specific field of the call-information record.

LINECALLINFOSTATE_CALLID
The call ID field of the call-information record.

LINECALLINFOSTATE_RELATEDCALLID
The related call ID field of the call-information record.

LINECALLINFOSTATE_ORIGIN
The origin field of the call-information record.

LINECALLINFOSTATE_REASON
The reason field of the call-information record.

LINECALLINFOSTATE_COMPLETIONID
The completion ID field of the call-information record.

LINECALLINFOSTATE_TRUNK
The trunk field of the call-information record.

LINECALLINFOSTATE_CALLERID
One of the callerID-related fields of the call-information record.

LINECALLINFOSTATE_CALLEDID
One of the calledID-related fields of the call-information record.

LINECALLINFOSTATE_CONNECTEDID
One of the cconnectedID-related fields of the call-information record.

LINECALLINFOSTATE_REDIRECTIONID
One of the redirectionID-related fields of the call-information record.

LINECALLINFOSTATE_REDIRECTINGID
One of the redirectingID-related fields of the call-information record.

LINECALLINFOSTATE_DISPLAY
The display field of the call-information record.
LINECALLINFOSTATE_USERUSERINFO
The user-to-user information of the call-information record.
LINECALLINFOSTATE_HIGHLEVELCOMP
The high level compatibility field of the call-information record.
LINECALLINFOSTATE_LOWLEVELCOMP
The low level compatibility field of the call-information record.
LINECALLINFOSTATE_CHARGINGINFO
The charging information of the call-information record.
LINECALLINFOSTATE_TERMINAL
The terminal mode information of the call-information record.
LINECALLINFOSTATE_DIALPARAMS
The dial parameters of the call-information record.
LINECALLINFOSTATE_MONITORMODES
One or more of the digit, tone, or media monitoring fields in the call-information record.

dwParam2

Unused.

dwParam3

Unused.

This message is sent to TAPI.DLL whenever the event occurs and TAPI.DLL has the line open. However, no **LINE_CALLINFO** messages are sent for a call after the call has entered the *idle* state.

LINE_CALLSTATE

[New - Windows 95]

```
htLine = (HTAPILINE) hLineDevice;
htCall = (HTAPICALL) hCallDevice;
dwMsg = (DWORD) LINE_CALLSTATE;
dwParam1 = (DWORD) LineCallState;
dwParam2 = (DWORD) StateData;
dwParam3 = (DWORD) MediaMode;
```

Sent to the **LINEEVENT** callback whenever the status of the specified call has changed. Several such messages will typically be sent during the lifetime of a call. The first such message for an incoming call will indicate the *offering* state. TAPI.DLL can use **TSPI_lineGetCallStatus** to find out more detailed information about the current status of the call.

htLine

Specifies TAPI.DLL's opaque object handle to the line device.

htCall

Specifies TAPI.DLL's opaque object handle to the call device.

dwMsg

The value **LINE_CALLSTATE**

dwParam1

Specifies the new call state. This parameter uses the following LINECALLSTATE_ constants:

LINECALLSTATE_IDLE
The call is idle—no call actually exists.

LINECALLSTATE_OFFERING
The call is being offered to the station, signaling the arrival of a new call. In some environments, a call in the *offering* state does not automatically alert the user; alerting is done by the switch instructing the line to ring, it does not affect any call states.

LINECALLSTATE_ACCEPTED
The call was offering and has been accepted. Note that in ISDN, the transition to the *accepted* state implicitly initiates altering to both parties.

LINECALLSTATE_DIALTONE
The call is receiving a dial tone from the switch, which means that the switch is ready to receive a dialed number.

LINECALLSTATE_DIALING
Destination address information (a phone number) is being sent to the switch via the call. Note that the operation **TSPI_lineGenerateDigits** does not place the line into the *dialing* state.

LINECALLSTATE_RINGBACK
The call is receiving ringback from the called address. Ringback indicates that the other station has been reached and is being alerted.

LINECALLSTATE_BUSY
The call is receiving a busy tone. Busy tone indicates that the call cannot be completed - either a circuit (trunk) or the remote party's station are in use.

LINECALLSTATE_SPECIALINFO
Special information is sent by the network. Special information is typically sent when the destination cannot be reached.

LINECALLSTATE_CONNECTED
The call has been established, the connection is made. Information is able to flow over the call between the originating address and the destination address.

LINECALLSTATE_PROCEEDING
Dialing has completed and the call is proceeding through the switch or telephone network.

LINECALLSTATE_ONHOLD
The call is on hold by the switch.

LINECALLSTATE_CONFERENCED
The call is currently a member of a multi-party conference call.

LINECALLSTATE_ONHOLDPENDCONF

The call is currently on hold while it is being added to a conference.

LINECALLSTATE_ONHOLDPENDTRANSFER
The call is currently on hold awaiting transfer to another number.

LINECALLSTATE_DISCONNECTED
The remote party has disconnected from the call.

LINECALLSTATE_UNKNOWN
The state of the call is not known. This may be due to limitations of the call progress detection implementation.

dwParam2

Specifies call-state-dependent information.

If *dwParam1* is **LINECALLSTATE_BUSY**, the *dwParam2* contains the details about the busy mode, and uses the following LINEBUSYMODE_ constants:

LINEBUSYMODE_STATION
The busy signal indicates that the called party's station is busy. This is usually signaled with a "normal" busy tone.

LINEBUSYMODE_TRUNK
The busy signal indicates that a trunk or circuit is busy. This is usually signaled with a "long" busy tone.

LINEBUSYMODE_UNKNOWN
The busy signal's specific mode is currently unknown, but may become known later.

LINEBUSYMODE_UNAVAIL
The busy signal's specific mode is unavailable and will not become known.

If *dwParam1* is LINECALLSTATE_DIALTONE, the *dwParam2* contains the details about the dialtone mode, and uses the following LINEDIALTONEMODE_ constants:

LINEDIALTONEMODE_NORMAL
This is a "normal" dialtone which typically is a continuous tone.

LINEDIALTONEMODE_SPECIAL
This is a special dialtone indicating a certain condition is currently in effect.

LINEDIALTONEMODE_INTERNAL
This an internal dialtone, as within a PBX.

LINEDIALTONEMODE_EXTERNAL
This is an external (public network) dialtone.

LINEDIALTONEMODE_UNKNOWN
The dialtone mode is currently known, but may become known later.

LINEDIALTONEMODE_UNAVAIL
The dialtone mode is unavailable and will not become known.

If *dwParam1* is LINECALLSTATE_SPECIALINFO, the *dwParam2* contains the details about the special info mode and uses the following LINESPECIALINFO_ constants:

LINESPECIALINFO_NOCIRCUIT
This special information tone preceeds a no circuit or emergency announcement (trunk blockage category).

LINESPECIALINFO_CUSTIRREG
This special information tone preceeds a vacant number, AIS, Centrex number change and non-working station, access code not dialed or dialed in error, manual intercept operator message (customer irregularity category).

LINESPECIALINFO_REORDER
This special information tone preceeds a reorder announcement (equipment irregularity category).

LINESPECIALINFO_UNKNOWN
Specifics about the special information tone are currently unknown but may become known later.

LINESPECIALINFO_UNAVAIL
Specifics about the special information tone are unavailable, and will not become known.

If *dwParam1* is LINECALLSTATE_DISCONNECTED, the *dwParam2* contains the details about the disconnect mode, and uses the following LINEDISCONNECTMODE_ constants:

LINEDISCONNECTMODE_NORMAL

This is a “normal” disconnect request by the remote party, the call was terminated normally.

LINEDISCONNECTMODE_UNKNOWN
The reason for the disconnect request is unknown.

LINEDISCONNECTMODE_REJECT
The remote user has rejected the call.

LINEDISCONNECTMODE_PICKUP
The call was picked up from elsewhere.

LINEDISCONNECTMODE_FORWARDED
The call was forwarded by the switch.

LINEDISCONNECTMODE_BUSY
The remote user’s station is busy.

LINEDISCONNECTMODE_NOANSWER
The remote user’s station does not answer.

LINEDISCONNECTMODE_BADADDRESS
The destination address in invalid.

LINEDISCONNECTMODE_CONGESTION
The network is congested.

LINEDISCONNECTMODE_INCOMPATIBLE
The remote user’s station equipment is incompatible for the type of call requested.

LINEDISCONNECTMODE_UNAVAIL
The remote user’s station equipment is incompatible for the type of call requested.

dwParam3

The media mode of the call, as far as it is known. This is a combination of LINEMEDIAMODE_ constants. If the service provider does not know the media mode, it should include the “UNKNOWN” bit together with all media modes currently being monitored for.

This message (with LINECALLSTATE_OFFERING) should be sent as the next message for an incoming call after **LINE_NEWCALL**. Other call state changes are reported whenever they occur; the message cannot be disabled.

The **LINE_CALLSTATE** message also notifies TAPI.DLL about the existence and state of outbound calls established as a side effect of other calls (for example, when an active call is put on hold and replaced by a new call in the *dialtone* state) or manually by the user (for example, on an attached phone device). The call state of such calls will reflect the actual state of the call, which will not be *offering*. By examining the call state, TAPI.DLL can determine whether the call is an inbound call that needs to answered.

The corresponding message at the TAPI level is used to inform applications of new incoming calls. This is not the case at the TSPI level; the **LINE_NEWCALL** message informs TAPI.DLL of new incoming calls. The **LINE_NEWCALL** message must precede this message.

The *dwParam3* parameter is used at the TAPI level to inform the recipient of the privilege level it has over the call.

LINE_CLOSE

[New - Windows 95]

```
htLine = (HTAPILINE) hLineDevice;
htCall = (HTAPICALL) 0;
dwMsg = (DWORD) LINE_CLOSE;
dwParam1 = (DWORD) 0;
dwParam2 = (DWORD) 0;
dwParam3 = (DWORD) 0;
```

Sent to the **LINEEVENT** callback when the specified line device has been forcibly closed. The line device handle or any call handles for calls on the line are no longer valid once this message has been sent. The service provider guarantees that all asynchronous requests on the line and all calls on the line have been reported complete by calling **ASYNC_COMPLETION** for each outstanding request before this message is sent. TAPI.DLL must not request any future operations using this line handle or its associated call handles.

htLine
Specifies TAPI.DLL's opaque object handle to the line device.

htCall
Unused

dwMsg
The value LINE_CLOSE

dwParam1
Unused.

dwParam2
Unused.

dwParam3
Unused.

This message is only sent to TAPI.DLL after an open line has been forcibly closed. This may be done, for example, when taking the line out of service or reconfiguring the service provider. Whether or not the line can be reopened immediately after a forced close is up to the service provider.

A line device may also be forcibly closed after the user has modified the configuration of that line or its driver. If the user wants the configuration changes to be effective immediately (as opposed to after the next system restart), and they affect the application's current view of the device (such as a change in device capabilities), a service provider may forcibly close the line device.

LINE_DEVSPECIFIC

[New - Windows 95]

```
htLine = (HTAPILINE) hLineDevice;
htCall = (HTAPICALL) 0;
dwMsg = (DWORD) LINE_DEVSPECIFIC;
dwParam1 = (DWORD) DeviceData1;
dwParam2 = (DWORD) DeviceData2;
dwParam3 = (DWORD) DeviceData3;
```

Sent to the **LINEEVENT** callback to notify TAPI.DLL about device-specific events occurring on a line or address. The meaning of the message and the interpretation of the *dwParam1* through *dwParam3* parameters is device specific.

htLine

Specifies TAPI.DLL's opaque object handle to the line device.

htCall

Unused

dwMsg

The value **LINE_DEVSPECIFIC**

dwParam1

Device specific.

dwParam2

Device specific.

dwParam3

Device specific.

This message is used by a service provider in conjunction with the **TSPI_lineDevSpecific** function. Its meaning is device specific.

TAPI.DLL sends the **LINE_DEVSPECIFIC** message to applications in response to receiving this message from a service provider. The *htLine* is translated to the appropriate *hLine* as the *hDevice* parameter at the TAPI level. The *dwParam1, dwParam2,* and *dwParam3* paramters are passed through unmodified.

This differs from the corresponding message at the TAPI level in that it is used only to report device-specific events occurring on a line or address. At the TSPI level, the **LINE_CALLDEVSPECIFIC** is used to report device-specific events happening on a call.

LINE_DEVSPECIFICFEATURE

[New - Windows 95]

```
htLine = (HTAPILINE) hLineDevice;
htCall = (HTAPICALL) 0;
dwMsg = (DWORD) LINE_DEVSPECIFICFEATURE;
dwParam1 = (DWORD) DeviceData1;
dwParam2 = (DWORD) DeviceData2;
dwParam3 = (DWORD) DeviceData3;
```

Sent to the **LINEEVENT** callback to notify TAPI.DLL about device-specific events occurring on a line or address. The meaning of the message and the interpretation of the *dwParam1* through *dwParam3* parameters is device specific.

htLine

Specifies TAPI.DLL's opaque object handle to the line device.

htCall

Unused

dwMsg

The value **LINE_DEVSPECIFICFEATURE**

dwParam1

Device specific.

dwParam2

Device specific.

dwParam3

Device specific.

This message is used by a service provider in conjunction with the **TSPI_lineDevSpecificFeature** function. Its meaning is device specific.

TAPI.DLL sends the **LINE_DEVSPECIFICFEATURE** message to applications in response to receiving this message from a service provider. The *htLine* is translated to the appropriate *hLine* as the *hDevice* parameter at the TAPI level. The *dwParam1, dwParam2,* and *dwParam3* paramters are passed through unmodified.

This differs from the corresponding message at the TAPI level in that it is used only to report device-specific feature events occurring on a line or address. At the TSPI level, the **LINE_CALLDEVSPECIFICFEATURE** is used to report device-specific feature events happening on a call.

LINE_GATHERDIGITS

[New - Windows 95]

```
htLine = (HTAPILINE) hLineDevice;
htCall = (HTAPICALL) hCallDevice;
dwMsg = (DWORD) LINE_GATHERDIGITS;
dwParam1 = (DWORD) LineGatherTerminate;
dwParam2 = (DWORD) dwEndToEndID;
dwParam3 = (DWORD) 0;
```

Sent to the **LINEEVENT** callback when the current buffered digit gathering request has terminated. This message is not sent when digit gathering is canceled. The digit buffer may be examined after this message has been received. The effect of examining the digit buffer before this message has been received is undefined.

htLine

Specifies TAPI.DLL's opaque object handle to the line device.

htCall

Specifies TAPI.DLL's opaque object handle to the call device.

dwMsg

The value **LINE_GATHERDIGITS**

dwParam1

Specifies the reason why digit gathering was terminated. This parameter uses the following LINEGATHERTERM_ constants:

LINEGATHERTERM_BUFFERFULL
: The requested number of digits has been gathered. The buffer is full.

LINEGATHERTERM_TERMDIGIT
: One of the termination digits matched a received digit. The matched termination digit is the last digit in the buffer.

LINEGATHERTERM_FIRSTTIMEOUT
: The first digit timeout expired. The buffer contains no digits.

LINEGATHERTERM_INTERTIMEOUT
: The inter digit timeout expired. The buffer contains at least one digit.

LINEGATHERTERM_CANCEL
: The request was canceled by a subsequent **TSPI_lineGatherDigits** request, or because the call terminated.

dwParam2

The *dwEndToEndID* that was specified in the original **TSPI_lineGatherDigits** request for which this is the final result.

dwParam3

Unused.

Calling **TSPI_lineGatherDigits** while digit gathering is in progress cancels the current digit gathering request and starts a new one, but does not result in the service provider sending this message.

The corresponding message at the TAPI level does not include an end-to-end ID. End-to-end marking is only done at the TSPI level.

LINE_GENERATE

[New - Windows 95]

```
htLine = (HTAPILINE) hLineDevice;
htCall = (HTAPICALL) hCallDevice;
dwMsg = (DWORD) LINE_GENERATE;
dwParam1 = (DWORD) LineGenerateTerminate;
dwParam2 = (DWORD) dwEndToEndID;
dwParam3 = (DWORD) 0;
```

Sent to the **LINEEVENT** callback to notify TAPI.DLL that the current digit or tone generation has terminated. Note that only one such generation request can be in progress on a given call at any time. This message is also sent when digit or tone generation is canceled.

htLine

Specifies TAPI.DLL's opaque object handle to the line device.

htCall

Specifies TAPI.DLL's opaque object handle to the call device.

dwMsg

The value **LINE_GENERATE**

dwParam1

Specifies the reason why digit or tone generation was terminated, and uses the following LINEGENERATETERM_ constants:

LINEGENERATETERM_DONE

The requested number of digits have been generated, or the requested tones have been generated for the requested duration.

LINEGENERATETERM_CANCEL

The digit or tone generation request was canceled by an invervening call to **TSPI_lineGenerateTone** or **TSPI_lineGenerateDigits**, or because the call terminated.

dwParam2

The *dwEndToEndID* parameter that was specified in the original **TSPI_lineGenerateDigits** or **TSPI_lineGenerateTone** request for which this is the final result.

dwParam3

Unused.

Calling either **TSPI_lineGenerateDigits** or **TSPI_lineGenerateTone** while digit or tone generation is in progress cancels the current digit or tone generation request and starts a new request and also results in the service provider sending this message.

The corresponding message at the TAPI level does not include an end-to-end ID. End-to-end marking is only done at the TSPI level.

LINE_LINEDEVSTATE

[New - Windows 95]

```
htLine = (HTAPILINE) hLineDevice;
htCall = (HTAPICALL) 0;
dwMsg = (DWORD) LINE_LINEDEVSTATE;
dwParam1 = (DWORD) LineDevState;
dwParam2 = (DWORD) DevStateData1;
dwParam3 = (DWORD) DevStateData2;
```

Sent to the **LINEEVENT** callback when the state of a line device has changed. TAPI.DLL can invoke **TSPI_lineGetLineDevStatus** to determine the new status of the line.

htLine

Specifies TAPI.DLL's opaque object handle to the line device.

htCall

Unused

dwMsg

The value **LINE_LINEDEVSTATE**
Specifies the callback instance supplied when opening the line.

dwParam1

Specifies the line device status item that has changed. This parameter can have multiple flags set and uses the following LINEDEVSTATE_ constants:

LINEDEVSTATE_OTHER
Device-status items other than those listed below have changed.

LINEDEVSTATE_RINGING
The switch tells the line to alert the user.

LINEDEVSTATE_CONNECTED
The line was previously disconnected and is now connected to the API.

LINEDEVSTATE_DISCONNECTED
This line was previously connected and is now disconnected from the API.

LINEDEVSTATE_MSGWAITON
The "message waiting" indicator is turned on.

LINEDEVSTATE_MSGWAITOFF
The "message waiting" indicator is turned off.

LINEDEVSTATE_INSERVICE
The line is connected to the API. This happens when the API is first activated, or when the line wire is physically plugged in and in service at the switch while the API is active.

LINEDEVSTATE_OUTOFSERVICE
The line is out of service at the switch or physically disconnected. The API cannot be used to operate on the line device.

LINEDEVSTATE_MAINTENANCE
Maintenance is being performed on the line at the switch. The API cannot be used to operate on the line device.

LINEDEVSTATE_NUMCALLS
The number of calls on the line device has changed.

LINEDEVSTATE_NUMCOMPLETIONS
The number of outstanding call completions on the line device has changed.

LINEDEVSTATE_TERMINALS
The terminal settings have changed.

LINEDEVSTATE_ROAMMODE
The roaming state of the line device has changed.

LINEDEVSTATE_BATTERY
The battery level has changed significantly (cellular).

LINEDEVSTATE_SIGNAL
The signal level has changed significantly (cellular).

LINEDEVSTATE_DEVSPECIFIC
The line's device-specific information has changed.

LINEDEVSTATE_REINIT
Items have changed in the configuration of line devices. Except in extreme cases, the service provider should use the normal configuration change notification mechanism as described in the "Overview" chapter instead of this message. Note that when *dwParam1* is LINEDEVSTATE_REINIT, *htLine* is unused, and should be set to NULL.

LINEDEVSTATE_LOCK
The locked status of the line device has changed.

dwParam2
The interpretation of this parameter depends on the value of *dwParam1*. If *dwParam1* is LINEDEVSTATE_RINGING, *dwParam2* contains the ring mode with which the switch instructs the line to ring. Valid ring modes are numbers in the range one to **dwNumRingModes**, where **dwNumRingModes** is a line device capability.

dwParam3
The interpretation of this parameter depends on the value of *dwParam1*. If *dwParam1* is LINEDEVSTATE_RINGING, *dwParam3* contains the ring count for this ring event. The ring count starts at zero.

The sending of this message can be controlled via **TSPI_lineSetStatusMessages**. The service provider must send **LINE_LINEDEVSTATE** messages for at least the set of status changes selected through that procedure. The service provider may send more than this set, however, it should try to limit its messages to this set for performance reasons. By default all status reporting will be disabled.

At the TSPI level, the service provider does not report state changes when the line is opened and closed, since there is only ever one Open outstanding for the device.

LINE_MONITORDIGITS

[New - Windows 95]

```
htLine = (HTAPILINE) hLineDevice;
htCall = (HTAPICALL) hCallDevice;
dwMsg = (DWORD) LINE_MONITORDIGITS;
dwParam1 = (DWORD) chDigit;
dwParam2 = (DWORD) LineDigitMode;
dwParam3 = (DWORD) 0;
```

Sent to the **LINEEVENT** callback whenever a digit is detected while digit monitoring is in progress. The sending of this message is controlled by the **TSPI_lineMonitorDigits** function.

htLine

Specifies TAPI.DLL's opaque object handle to the line device.

htCall

Specifies TAPI.DLL's opaque object handle to the call device.

dwMsg

The value **LINE_MONITORDIGITS**

dwParam1

The low-order byte contains the last digit received in ASCII.

dwParam2

Specifies the digit mode that was detected. This parameter uses the following LINEDIGITMODE_ constants:

LINEDIGITMODE_PULSE

Detect digits as audible clicks that are the result of rotary pulse sequences. Valid digits for pulse are '0' through '9'.

LINEDIGITMODE_DTMF

Detect digits as DTMF tones. Valid digits for DTMF are '0' through '9', 'A', 'B', 'C', 'D', '*', and '#'.

LINEDIGITMODE_DTMFEND

Detect and provide notification of DTMF down edges. Valid digits for DTMF are '0' through '9', 'A', 'B', 'C', 'D', '*', and '#'.

dwParam3

Unused.

This message is sent only when digit monitoring is enabled.

LINE_MONITORMEDIA

[New - Windows 95]

```
htLine = (HTAPILINE) hLineDevice;
htCall = (HTAPICALL) hCallDevice;
dwMsg = (DWORD) LINE_MONITORDIGITS;
dwParam1 = (DWORD) LineMediaMode;
dwParam2 = (DWORD) 0;
dwParam3 = (DWORD) 0;
```

Sent to the **LINEEVENT** callback whenever a change in the call's media mode is detected and media mode monitoring is enabled. The sending of this message is controlled by the **TSPI_lineMonitorMedia** function.

htLine

Specifies TAPI.DLL's opaque object handle to the line device.

htCall

Specifies TAPI.DLL's opaque object handle to the call device.

dwMsg

The value **LINE_MONITORMEDIA**

dwParam1

Specifies the new media mode. This parameter uses the following LINEMEDIAMODE_ constants:

LINEMEDIAMODE_INTERACTIVEVOICE
: The presence of voice energy on the call, and the call is treated as an interactive call with humans on both ends.

LINEMEDIAMODE_AUTOMATEDVOICE
: The presence of voice energy on the call and the voice is locally handled by an automated application.

LINEMEDIAMODE_DATAMODEM
: A data modem session on the call.

LINEMEDIAMODE_G3FAX
: A group 3 fax is being sent or received over the call.

LINEMEDIAMODE_TDD
: A TDD session on the call. TDD stands for Telephony Devices for the Deaf.

LINEMEDIAMODE_G4FAX
: A group 4 fax is being sent or received over the call.

LINEMEDIAMODE_DIGITALDATA
: Digital data being sent or received over the call.

LINEMEDIAMODE_TELETEX
: A teletex session on the call. Teletex is one of the telematic services.

LINEMEDIAMODE_VIDEOTEX
: A videotex session on the call. Videotex is one of the telematic services.

LINEMEDIAMODE_TELEX
: A telex session on the call. Telex is one of the telematic services.

LINEMEDIAMODE_MIXED
: A mixed session on the call. Mixed is one of the telematic services.

LINEMEDIAMODE_ADSI
: An ADSI session on the call. ADSI stands for Analog Display Services Interface.

dwParam2

Unused.

dwParam3

Unused.

This message is sent only when media mode monitoring is enabled.

LINE_MONITORTONE

[New - Windows 95]

```
htLine = (HTAPILINE) hLineDevice;
htCall = (HTAPICALL) hCallDevice;
dwMsg = (DWORD) LINE_MONITORTONE;
dwParam1 = (DWORD) dwAppSpecific;
dwParam2 = (DWORD) dwToneListID;
dwParam3 = (DWORD) 0;
```

Sent to the **LINEEVENT** callback whenever a monitored tone is detected. The sending of this message is controlled by the **TSPI_lineMonitorTones** function. A single tone detection may result in the service provider sending this message several times in the case where the detected tone matches several tone descriptions in several different tone lists. A tone may “match” several non-identical tone descriptions depending on how close they are to one another and the discrimination limits achieved by the service provider. Multiple matches may occur within the same or different tone lists.

htLine

Specifies TAPI.DLL’s opaque object handle to the line device.

htCall

Specifies TAPI.DLL’s opaque object handle to the call device.

dwMsg

The value **LINE_MONITORTONE**

dwParam1

Specifies the application-specific field **dwAppSpecific** of the **LINEMONITORTONE** structure for the tone that was detected.

dwParam2

Specifies the **dwToneListID** of the tone list containing a matching tone description.

dwParam3

Unused.

This message is sent once for each tone it matches in each tone list. Since there can be multiple matches both within a single tone list and across multiple tone lists, a single tone detection may result in several messages, indicating the same **dwToneListID** and several messages with a differing **dwToneListID**.

The *dwParam2* field in the corresponding message at the TAPI level is unused. TAPI.DLL forwards all tone monitoring requests from all applications to the service provider to be handled concurrently. It assigns a unique **dwToneListID** to each tone list and uses the **dwToneListID** added to *dwParam2* at the TAPI level to discriminate the resulting messages.

LINE_NEWCALL

[New - Windows 95]

```
htLine = (HTAPILINE) hLineDevice;
htCall = (HTAPICALL) 0;
dwMsg = (DWORD) LINE_NEWCALL;
dwParam1 = (DWORD)(HDRVCALL) hdCall;
dwParam2 = (DWORD)(LPHTAPICALL) &htCall;
dwParam3 = (DWORD) 0;
```

Sent to the **LINEEVENT** callback whenever a new call that TAPI.DLL has not originated arrives on a line that TAPI.DLL has open. This must be the first message sent regarding that call. TAPI.DLL writes the *htCall* opaque handle to the location passed by the service provider as *dwParam2*. This gives the service provider the *htCall* value to be used in subsequent messages.

htLine

Specifies TAPI.DLL's opaque object handle to the line device.

htCall

Unused

dwMsg

The value **LINE_NEWCALL**

dwParam1

Specifies the service provider's opaque handle for the call, of type HDRVCALL. TAPI.DLL will pass this value as the *hdCall* parameter to identify the call in subsequent procedures it invokes to operate on the call.

dwParam2

A pointer of type LPHTAPICALL pointing to a HTAPICALL. TAPI.DLL writes TAPI.DLL's opaque handle for the call to the indicated location. The service provider must save this value and pass it as the *htCall* parameter to identify the call in subsequent events it reports for the call.

This parameter can also acquire a value of NULL. This case is described in the Comments section.

dwParam3

Unused.

The service provider should send the **LINE_CALLSTATE** message as the next message for this call. The **LINE_NEWCALL** event is unusual in that it also passes a value back to the service provider.

This function reports any "new" calls that originate in the service provider (inbound, outbound, initiated at the phone, and so on) for which TAPI.DLL and the service provider have not yet exchanged opaque handles. The handles are exchanged so that TAPI.DLL and the service provider can subsequently make requests and report events involving the call. Since these new calls are not necessarily inbound, the calls can initially be in *any* state, not necessarily the *offering* state. If the service provider starts up and discovers that one or more calls are already active on the line, it will inform TAPI.DLL of them with **LINE_NEWCALL** messages followed by **LINE_CALLSTATE** messages indicating the current state. A new outgoing call, initiated on the phone by the user, would be reported with a **LINE_NEWCALL** message, and the initial **LINE_CALLSTATE** message would indicate that the call was in **DIALTONE** state (and then continuing on from there).

If the service provider passes a large number of calls to TAPI.DLL in a very short amount of time (during the same interrupt cycle), TAPI.DLL can become backlogged in processing those calls. When this happens, TAPI.DLL signals to the service provider that it should wait a short time before sending any more calls. It signals this by writing a value of NULL, instead of a valid HTAPICALL, into the location pointed to by the *dwParam2* parameter of **LINE_NEWCALL**. This indicates that the attempt to process the newly offered call handle was not succesful, most likely due to a temporary inability to allocate memory. The service provider may respond by dropping the call or by resending the **LINE_NEWCALL** message after a scheduling delay (during which time the service provider should yield the processor to allow TAPI.DLL to process other pending actions). In any case, no further messages regarding the new call may be passed to TAPI.DLL until the handle exchange succeeds. When the location pointed to by *dwParam2* acquires a non-null value, the service provider knows that this value is a valid HTAPICALL handle to the call.

There is no directly corresponding message at the TAPI level. This message is used at the TSPI level to uniquely and unambiguously introduce a new incoming call to TAPI.DLL and retrieve TAPI.DLL's opaque identifier for the call.

TSPI Data Types

TSPI Data Types

About TSPI Data Types

This chapter describes data types used by the Telephony SPI.

Reference

DRV_REQUESTID

[New - Windows 95]

This type is used to supply a unique identifier for a request to the service provider. A value of this type is passed as a parameter to every function that allows for asynchronous operation. If the operation is asynchronous, the service provider returns this value as the return value of the function. Whenever the service provider flags a request as asynchronous in this way, it must eventually report the operation complete by calling the **ASYNC_COMPLETION** callback function.

TAPI.DLL guarantees that DRV_REQUESTID values it supplies are strictly positive, that is, between the values of 0x00000001 and 0x7FFFFFF, inclusive. Furthermore, the values are “unique” in the sense that no value returned from a function to flag the request as asynchronous will be re-used before the operation is reported complete.

HDRVCALL

[New - Windows 95]

This type represents a service provider's opaque handle for a call data structure. It is the responsibility of the service provider to be able to resolve a value of this type into a reference to the appropriate data structure instance.

HDRVLINE

[New - Windows 95]

This type represents a service provider’s opaque handle for a line data structure. It is the responsibility of the service provider to resolve a value of this type into a reference to the appropriate data structure instance.

HDRVPHONE

[New - Windows 95]

This type represents a service provider’s opaque handle for a phone data structure. It is the responsibility of the service provider to resolve a value of this type into a reference to the appropriate data structure instance.

HTAPICALL

[New - Windows 95]

This type represents TAPI.DLL's opaque handle for a call data structure. It is the responsibility of TAPI.DLL to resolve a value of this type into a reference to the appropriate data structure instance. The service provider should not attempt to reference through this as if it were a pointer, make assumptions about its values, or interpret its representation in any way other than passing its value to TAPI.DLL at appropriate times.

HTAPILINE

[New - Windows 95]

This type represents TAPI.DLL's opaque handle for a line data structure. It is the responsibility of TAPI.DLL to resolve a value of this type into a reference to the appropriate data structure instance. The service provider should not attempt to reference through this as if it were a pointer, make assumptions about its values, or interpret its representation in any way other than passing its value to TAPI.DLL at appropriate times.

HTAPIPHONE

[New - Windows 95]

This type represents TAPI.DLL's opaque handle for a phone data structure. It is the responsibility of TAPI.DLL to resolve a value of this type into a reference to the appropriate data structure instance. The service provider should not attempt to reference through this as if it were a pointer, make assumptions about its values, or interpret its representation in any way other than passing its value to TAPI.DLL at appropriate times.

INITIALIZE_NEGOTIATION

[New - Windows 95]

This is a special value of type DWORD. It may be passed as the *dwDeviceID* in the **TSPI_lineNegotiateTSPIVersion** and **TSPI_phoneNegotiateTSPIVersion** functions. Doing so indicates that TAPI.DLL wishes to negotiate a TSPI interface version number independent of any specific devices.

TSPIMessage

[New - Windows 95]

This is an enumeration type that defines the set of additional **LINEEVENT** and **PHONEEVENT** messages appearing in the TSPI that do not appear in the TAPI.

Values

The following **TSPIMessage** values are defined by the TSPI:

TSPI_MESSAGE_BASE
Specifies the lowest number in the range of TSPI-specific message values. This has no meaning by itself, it simply serves as a base for defining the other values.

LINE_NEWCALL
Specifies that a new, incoming call has arrived and introduces it to TAPI.DLL. This must be the first message sent to TAPI.DLL for a new incoming call. TAPI.DLL returns its opaque identifier for the call as part of its handling of this message.

LINE_CALLDEVSPECIFIC
Specifies that a device-specific event has occurred on a call device.

LINE_CALLDEVSPECIFICFEATURE
Specifies that a device-specific feature event has occurred on a call device.

Line Device Structures

Line Device Structures

Line Device Structures

About Line Device Structures

Introdcution

The data structures described in this chapter are identical to those defined at the TAPI level. The TSPI header file does not define them. It includes them from the TAPI header file. The following sections defining the line-related and phone-related data structures are identical to the corresponding desctiptions in the TAPI interface.

In the case of most of the larger data structures, the responsibility for filling in fields is divided between the service provider and TAPI.DLL. The service provider must preserve the values present in fields “owned” by TAPI.DLL. The description of which fields must be set by the service provider and which fields must be preserved is provided in the “Functions” chapter in the functions that refer to that data structure.

For each structure, whether it is defined by the TSPI or by the TAPI, this chapter lists the following items:

- The purpose of the structure
- A description of the values or fields
- A description of the structure’s extensibility
- Optional comments about using the structure
- Optional references to other functions, messages, constants, or structures.

Memory for all data structures whose representation is published and shared by both TAPI.DLL and the service provider is allocated by TAPI.DLL or an application using TAPI.DLL. TAPI.DLL passes a far pointer to the TSPI function that returns the information. The TSPI fills the data structure with the requested information. If the operation is asynchronous, the information is not available until the asynchronous reply callback indicates success.

Opaque Handles

A few types defined by the TSPI are “opaque handles”. These are used in the TSPI as public references to private data structures. This allows type-checking of parameters supplied to the interface procedures while still giving a measure of type protection. Only the “owner” of the private data structure knows how to interpret the opaque type as a reference to its data structure representation. As an example of how opaque handles are used, consider a phone device. Both TAPI.DLL and the service provider typically maintain data structures representing their respective views of the device.

Typical phone data structures maintained by TAPI.DLL and a service provider are shown. Each contains an opaque handle for the other data structure (represented in the graphic by the solid circles). These were exchanged at an early initialization step. When TAPI.DLL calls a function in the TSPI interface, it passes the opaque handle to refer to the device. The service provider knows how to resolve this as a reference (arrow) to its data structure. A similar process occurs when the service provider calls a callback function in TAPI.DLL.

Opaque Handles and Private Data Structures

Opaque handles are used in the TSPI to refer to the data structures representing lines, phones and call appearances. These appear as parameters to most of the functions and callbacks. There are few functions that refer to the device using a device ID instead of an opaque handle. In such a case, the reference is to the device itself rather than to a data structure. Functions that work in this fashion are typically those called at early initialization times before data structures have been created and opaque handles have been exchanged.

It is up to the service provider to decide how to interpret these handles. A service provider typically uses the pointer to its data structure or to the index into an array of data structures as its opaque handle. The only restriction is that the service provider may not use the value zero as an HDRVLINE, HDRVCALL, or HDRVPHONE. The value zero or NULL for a handle has a special meaning in some operations.

Extensibility

Provisions are made for extending constants and structures both in a device independent way and in a device-specific (vendor-specific) way.

In constants that are scalar enumerations, a range of values is reserved for future common extensions. The remainder of values is identified as device specific. A vendor can define meanings for these values in any way desired. The interpretation of these values is keyed to the *extension ID* provided via the **LINEDEVCAPS** data structure. For constants that are defined as bit flags, a range of low-order bits are reserved, where the high-order bits can be extension specific. It is recommended that extended values and bit arrays use bits from the highest value or high-order bit down. This leaves the option to move the border between the common portion and extension portion if there is need to do so in the future. Extensions to data structures are assigned a variably sized field with size/offset being part of the fixed part. The TSPI describes for each data structures what device-specific extensions are allowed.

In addition to recognizing a specific extension ID, TAPI.DLL (operating on behalf of an application) must negotiate the extension version number that the application and the service provider will operate under. This is done using the **TSPI_lineNegotiateExtVersion** and **TSPI_phoneNegotiateExtVersion** functions.

An extension ID is a globally unique identifier. There is no central registry for extension IDs. Instead, they are generated locally by the manufacturer by a utility that is available with the toolkit. The number is made up parts such as a (unique) LAN address, time of day, random number, to guarantee global uniqueness. Globally Unique Identifiers are designed to be distinguishable from HP/DEC universally unique identifiers and are thus fully compatible with them.

Reference

LINEADDRESSCAPS

[New - Windows 95]

```
typedef struct lineaddresscaps_tag {
  DWORD  dwTotalSize;
  DWORD  dwNeededSize;
  DWORD  dwUsedSize;

  DWORD  dwLineDeviceID;

  DWORD  dwAddressSize;
  DWORD  dwAddressOffset;

  DWORD  dwDevSpecificSize;
  DWORD  dwDevSpecificOffset;

  DWORD  dwAddressSharing;
  DWORD  dwAddressStates;
  DWORD  dwCallInfoStates;
  DWORD  dwCallerIDFlags;
  DWORD  dwCalledIDFlags;
  DWORD  dwConnectedIDFlags;
  DWORD  dwRedirectionIDFlags;
  DWORD  dwRedirectingIDFlags;
  DWORD  dwCallStates;
  DWORD  dwDialToneModes;
  DWORD  dwBusyModes;
  DWORD  dwSpecialInfo;
  DWORD  dwDisconnectModes;

  DWORD  dwMaxNumActiveCalls;
  DWORD  dwMaxNumOnHoldCalls;
  DWORD  dwMaxNumOnHoldPendingCalls;
  DWORD  dwMaxNumConference;
  DWORD  dwMaxNumTransConf;

  DWORD  dwAddrCapFlags;
  DWORD  dwCallFeatures;
  DWORD  dwRemoveFromConfCaps;
  DWORD  dwRemoveFromConfState;
  DWORD  dwTransferModes;
  DWORD  dwParkModes;

  DWORD  dwForwardModes;
  DWORD  dwMaxForwardEntries;
  DWORD  dwMaxSpecificEntries;
  DWORD  dwMinFwdNumRings;
  DWORD  dwMaxFwdNumRings;

  DWORD  dwMaxCallCompletions;
  DWORD  dwCallCompletionConds;
  DWORD  dwCallCompletionModes;
  DWORD  dwNumCompletionMessages;
  DWORD  dwCompletionMsgTextEntrySize;
  DWORD  dwCompletionMsgTextSize;
  DWORD  dwCompletionMsgTextOffset;
} LINEADDRESSCAPS, FAR *LPLINEADDRESSCAPS;
```

The **LINEADDRESSCAPS** structure describes the capabilities of a specified address.

dwTotalSize
The total size in bytes allocated to this data structure.

dwNeededSize
The size in bytes for this data structure that is needed to hold all the returned information.

dwUsedSize
The size in bytes of the portion of this data structure that contains useful information.

dwLineDeviceID
Specifies the device ID of the line device with which this address is associated.

dwAddressSize
dwAddressOffset
The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized address field.

dwDevSpecificSize
dwDevSpecificOffset
The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device-specific field.

dwAddressSharing
Specifies the sharing mode of the address. Values are:

LINEADDRESSSHARING_PRIVATE
An address with *private* sharing mode is only assigned to a single line or station.

LINEADDRESSSHARING_BRIDGEDEXCL
An address with a *bridged-exclusive* sharing mode is assigned to one or more other lines or stations. The *exclusive* portion refers to the fact that only one of the bridged parties can be connected with a remote party at any given time.

LINEADDRESSSHARING_BRIDGEDNEW
An address with a *bridged-new* sharing mode is assigned to one or more other lines or stations. The *new* portion refers to the fact that activities by the different bridged parties result in the creation of new calls on the address.

LINEADDRESSSHARING_BRIDGEDSHARED
An address with a *bridged-shared* sharing mode is also assigned to one or more other lines or stations. The *shared* portion refers to the fact that if one of the bridged parties is connected with a remote party, the remaining bridged parties can share in the conversation (as in a conference) by activating that call appearance.

LINEADDRESSSHARING_MONITORED
An address with a monitored address mode simply monitors the status of that address. The status is either *idle* or *in use*. The message **LINE_ADDRESSSTATE** notifies the application about these changes.

dwAddressStates
This field contains the address state changes for which the application may get notified in the **LINE_ADDRESSSTATE** callback message. This parameter uses the following LINEADDRESSSTATE_ constants:

LINEADDRESSSTATE_OTHER
Address-status items other than those listed below have changed.

LINEADDRESSSTATE_DEVSPECIFIC
The device-specific item of the address status has changed.

LINEADDRESSSTATE_INUSEZERO
The address has changed to idle (it is now in use by zero stations).

LINEADDRESSSTATE_INUSEONE
The address has changed from being idle or from being in use by many bridged stations to being in use by just one station.

LINEADDRESSSTATE_INUSEMANY
The monitored or bridged address has changed to being in use by one station to being used by more than one station.

LINEADDRESSSTATE_NUMCALLS
The number of calls on the address has changed. This is the result of events such as a new

inbound call, an outbound call on the address, or a call changing its hold status.

LINEADDRESSSTATE_FORWARD
The forwarding status of the address has changed, including the number of rings for determining a no answer condition. The application should check the address status to determine details about the address's current forwarding status.

LINEADDRESSSTATE_TERMINALS
The terminal settings for the address have changed.

dwCallInfoStates

This field describes the call info elements that are meaningful for all calls on this address, and uses the LINECALLINFOSTATE_ constants. An application may get notified about changes in some of these states in **LINE_CALLINFO** messages. Values are:

LINECALLINFOSTATE_OTHER
Call information items other than those listed below have changed.

LINECALLINFOSTATE_DEVSPECIFIC
The device-specific field of the call-information record.

LINECALLINFOSTATE_BEARERMODE
The bearer mode field of the call-information record.

LINECALLINFOSTATE_RATE
The rate field of the call-information record.

LINECALLINFOSTATE_MEDIAMODE
The media mode field of the call-information record.

LINECALLINFOSTATE_APPSPECIFIC
The application-specific field of the call-information record.

LINECALLINFOSTATE_CALLID
The call ID field of the call-information record.

LINECALLINFOSTATE_RELATEDCALLID
The related call ID field of the call-information record.

LINECALLINFOSTATE_ORIGIN
The origin field of the call-information record.

LINECALLINFOSTATE_REASON
The reason field of the call-information record.

LINECALLINFOSTATE_COMPLETIONID
The completion ID field of the call-information record.

LINECALLINFOSTATE_NUMOWNERINCR
The number of owner field in the call-information record was increased.

LINECALLINFOSTATE_NUMOWNERDECR
The number of owner field in the call-information record was decreased.

LINECALLINFOSTATE_NUMMONITORS
The number of monitors field in the call-information record has changed.

LINECALLINFOSTATE_TRUNK
The trunk field of the call-information record.

LINECALLINFOSTATE_CALLERID
One of the callerID-related fields of the call-information record.

LINECALLINFOSTATE_CALLEDID
One of the calledID-related fields of the call-information record.

LINECALLINFOSTATE_CONNECTEDID
One of the connectedID-related fields of the call-information record.

LINECALLINFOSTATE_REDIRECTIONID
One of the redirectionID-related fields of the call-information record.

LINECALLINFOSTATE_REDIRECTINGID
One of the redirectingID-related fields of the call-information record.

LINECALLINFOSTATE_DISPLAY
The display field of the call-information record.

LINECALLINFOSTATE_USERUSERINFO
The user-to-user information of the call-information record.

LINECALLINFOSTATE_HIGHLEVELCOMP
The high level compatibility field of the call-information record.
LINECALLINFOSTATE_LOWLEVELCOMP
The low level compatibility field of the call-information record.
LINECALLINFOSTATE_CHARGINGINFO
The charging information of the call-information record.
LINECALLINFOSTATE_TERMINAL
The terminal mode information of the call-information record.
LINECALLINFOSTATE_DIALPARAMS
The dial parameters of the call-information record.
LINECALLINFOSTATE_MONITORMODES
One or more of the digit, tone, or media monitoring fields in the call-information record.

dwCallerIDFlags
dwCalledIDFlags
dwConnectedIDFlags
dwRedirectionIDFlags
dwRedirectingIDFlags

These fields describe the various kinds of party ID information that may be provided for calls on this address. The following LINECALLPARTYID_ constants are used:

LINECALLPARTYID_BLOCKED
Caller ID information for the call has been blocked by the caller, but would otherwise have been available.
LINECALLPARTYID_OUTOFAREA
Caller ID information for the call is not available as it is not propagated all the way by the network.
LINECALLPARTYID_NAME
The caller ID information for the call is the caller's name (from a table maintained inside the switch). It is provided in the caller ID name variably sized field.
LINECALLPARTYID_ADDRESS
The caller ID information for the call is the caller's number, and is provided in the caller ID variably sized field.
LINECALLPARTYID_PARTIAL
Caller ID information for the call is valid, but is limited to partial number information.
LINECALLPARTYID_UNKNOWN
Caller ID information is currently unknown, but it may become known later.
LINECALLPARTYID_UNAVAIL
Caller ID information is unavailable and will not become known later.

dwCallStates

This field describes the various call states that can be reported for calls on this address. This field uses the following LINECALLSTATE_ constants:

LINECALLSTATE_IDLE
The call is *idle*—no call actually exists.
LINECALLSTATE_OFFERING
The call is being offered to the station, signaling the arrival of a new call. In some environments, a call in the *offering* state does not automatically alert the user; alerting is done by the switch instructing the line to ring, it does not affect any call states.
LINECALLSTATE_ACCEPTED
The call was offering and has been accepted. This indicates to other (monitoring) applications that the current owner application has claimed responsibility for answering the call. In ISDN, this also initiates alerting to both parties.
LINECALLSTATE_DIALTONE
The call is receiving a dial tone from the switch, which means that the switch is ready to receive a dialed number.
LINECALLSTATE_DIALING
Destination address information (a phone number) is being sent to the switch over the call. Note that the operation **TSPI_lineGenerateDigits** does not place the line into the *dialing* state.
LINECALLSTATE_RINGBACK

The call is receiving ringback from the called address. Ringback indicates that the other station has been reached and is being alerted.

LINECALLSTATE_BUSY
The call is receiving a busy tone. Busy tone indicates that the call cannot be completed because either a circuit (trunk) or the remote party's station is in use.

LINECALLSTATE_SPECIALINFO
Special information is sent by the network. Special information is typically sent when the destination cannot be reached.

LINECALLSTATE_CONNECTED
The call has been established, the connection is made. Information is able to flow over the call between the originating address and the destination address.

LINECALLSTATE_PROCEEDING
Dialing has completed and the call is proceeding through the switch or telephone network.

LINECALLSTATE_ONHOLD
The call is on hold by the switch.

LINECALLSTATE_CONFERENCED
The call is currently a member of a multi-party conference call.

LINECALLSTATE_ONHOLDPENDCONF
The call is currently on hold while it is being added to a conference.

LINECALLSTATE_ONHOLDPENDTRANSFER
The call is currently on hold awaiting transfer to another number.

LINECALLSTATE_DISCONNECTED
The remote party has disconnected from the call.

LINECALLSTATE_UNKNOWN
The state of the call is not known. This may be due to limitations of the call progress detection implementation.

dwDialToneModes

This field describes the various dialtone modes that can possibly be reported for calls made on this address and uses the following LINEDIALTONEMODE_ constants. This field is only meaningful if the *dialtone* call state can be reported.

LINEDIALTONEMODE_NORMAL
This is a "normal" dialtone which typically is a continuous tone.

LINEDIALTONEMODE_SPECIAL
This is a special dialtone indicating a certain condition is currently in effect.

LINEDIALTONEMODE_INTERNAL
This an internal dialtone, as within a PBX.

LINEDIALTONEMODE_EXTERNAL
This is an external (public network) dialtone.

LINEDIALTONEMODE_UNKNOWN
The dialtone mode is currently unknown, but may become known later.

LINEDIALTONEMODE_UNAVAIL
The dialtone mode is unavailable and will not become known.

dwBusyModes

This field describes the various busy modes that can possibly be reported for calls made on this address, and uses the following LINEBUSYMODE_ constants. This field is only meaningful of the *busy* call state can be reported.

LINEBUSYMODE_STATION
The busy signal indicates that the called party's station is busy. This is usually signaled by means of a "normal" busy tone.

LINEBUSYMODE_TRUNK
The busy signal indicates that a trunk or circuit is busy. This is usually signaled by means of a "long" busy tone.

LINEBUSYMODE_UNKNOWN
The busy signal's specific mode is currently unknown, but may become known later.

LINEBUSYMODE_UNAVAIL
The busy signal's specific mode is unavailable and will not become known.

dwSpecialInfo

This field describes the various kinds of special information that can be reported for calls made on this address. It uses the following LINESPECIALINFO_ constants. This field is only meaningful of the *specialInfo* call state can be reported.

LINESPECIALINFO_NOCIRCUIT
: This special information tone preceeds a no circuit or emergency announcement (trunk blockage category).

LINESPECIALINFO_CUSTIRREG
: This special information tone preceeds a vacant number, AIS, Centrex number change and non-working station, access code not dialed or dialed in error, manual intercept operator message (customer irregularity category).

LINESPECIALINFO_REORDER
: This special information tone preceeds a reorder announcement (equipment irregularity category).

LINESPECIALINFO_UNKNOWN
: Specific about the special information tone are currently unknown but may become known later.

LINESPECIALINFO_UNAVAIL
: Specifics about the special information tone are unavailable, and will not become known.

dwDisconnectModes

This field describes the various disconnect modes that can possibly be reported for calls made on this address, and uses the LINEDISCONNECTMODE_ constants listed below. This field is only meaningful of the *disconnected* call state can be reported. Values are:

LINEDISCONNECTMODE_NORMAL
: This is a “normal” disconnect request by the remote party, the call was terminated normally.

LINEDISCONNECTMODE_UNKNOWN
: The reason for the disconnect request is unknown.

LINEDISCONNECTMODE_REJECT
: The remote user has rejected the call.

LINEDISCONNECTMODE_PICKUP
: The call was picked up from elsewhere.

LINEDISCONNECTMODE_FORWARDED
: The call was forwarded by the switch.

LINEDISCONNECTMODE_BUSY
: The remote user’s station is busy.

LINEDISCONNECTMODE_NOANSWER
: The remote user’s station does not answer.

LINEDISCONNECTMODE_BADADDRESS
: The destination address in invalid.

LINEDISCONNECTMODE_CONGESTION
: The network is congested.

LINEDISCONNECTMODE_INCOMPATIBLE
: The remote user’s station equipment is incompatible for the type of call requested.

LINEDISCONNECTMODE_UNAVAIL
: The remote user’s station equipment is incompatible for the type of call requested.

dwMaxNumActiveCalls

This field contains the maximum number of active call appearances that the address can handle. This number does not include calls on hold or on hold pending transfer or conference.

dwMaxNumOnHoldCalls

This field contains the maximum number of call appearances at the address that can be on hold.

dwMaxNumOnHoldPendingCalls

This field contains the maximum number of call appearances at the address that can be on hold pending transfer or conference.

dwMaxNumConference

This field contains the maximum number of parties that can be conferenced in a single conference call on this address.

dwMaxNumTransConf

This field specifies the number of parties (including “self”) that can be added in a conference call that is initiated as a generic consultation call using the **lineSetupTransfer** function.

dwAddrCapFlags

This field contains a series of packed bit flags that describe a variety of address capabilities. It uses the following LINEADDRCAPFLAGS_ constants:

LINEADDRCAPFLAGS_FWDNUMRINGS
Specifies whether the number of rings for a no answer can be specified when forwarding calls on no answer.

LINEADDRCAPFLAGS_PICKUPGROUPID
Specifies whether a group ID is required for call pickup.

LINEADDRCAPFLAGS_SECURE
Specifies whether calls on this address can be made secure at call setup time.

LINEADDRCAPFLAGS_BLOCKIDDEFAULT
Specifies whether the network by default sends or blocks caller ID information when making a call on this address. If TRUE, ID information is blocked by default; if FALSE, ID information is transmitted by default.

LINEADDRCAPFLAGS_BLOCKIDOVERRIDE
Specifies whether the default setting for sending or blocking of caller ID information can be overridden per call. If TRUE, override is possible; if FALSE, override is not possible.

LINEADDRCAPFLAGS_DIALED
Specifies whether an destination address can be dialed on this address for making a call. TRUE if a destination address must be dialed; FALSE if the destination address is fixed (as with a “hot phone”).

LINEADDRCAPFLAGS_ORIGOFFHOOK
Specifies whether the originating party’s phone can automatically be taken offhook when making calls.

LINEADDRCAPFLAGS_DESTOFFHOOK
Specifies whether the called party’s phone can automatically be forced offhook when making calls.

LINEADDRCAPFLAGS_FWDCONSULT
Specifies whether call forwarding involves the establishment of a consultation call.

LINEADDRCAPFLAGS_SETUPCONFNULL
Specifies whether setting up a conference call starts out with an initial call (FALSE) or with no initial call (TRUE).

LINEADDRCAPFLAGS_AUTORECONNECT
Specifies whether dropping a consultation call automatically reconnects to the call on consultation hold. TRUE if reconnect happens automatically, FALSE otherwise.

LINEADDRCAPFLAGS_COMPLETIONID
Specifies whether the completion IDs returned by **TSPI_lineCompleteCall** are useful and unique. TRUE is useful; FALSE otherwise.

LINEADDRCAPFLAGS_TRANSFERHELD
Specifies whether a (hard) held call can be transferred. Often, only calls on consultation hold may be able to be transferred.

LINEADDRCAPFLAGS_TRANSFERMAKE
Specifies whether an entirely new call can be established for use as a consultation call on transfer.

LINEADDRCAPFLAGS_CONFERENCEHELD
Specifies whether a (hard) held call can be conferenced to. Often, only calls on consultation hold may be able to be added to as a conference call.

LINEADDRCAPFLAGS_CONFERENCEMAKE
Specifies whether an entirely new call can be established for use as a consultation call (to add) on conference.

LINEADDRCAPFLAGS_PARTIALDIAL
Specifies whether partial dialing is available.

LINEADDRCAPFLAGS_FWDSTATUSVALID

Specifies whether the forwarding status in the **LINEADDRESSSTATUS** structure for this address is valid.

LINEADDRCAPFLAGS_FWDINTEXTADDR
Specifies whether internal and external calls can be forwarded to different forwarding addresses. This flag is only meaningful if forwarding of internal and external calls can be controlled separately. This flag is TRUE if internal and external calls can be forwarded to different destination addresses; FALSE otherwise.

LINEADDRCAPFLAGS_FWDBUSYNAADDR
Specifies whether call forwarding for busy and no answer can use different forwarding addresses. This flag is only meaningful if forwarding for busy and no answer can be controlled separately. This flag is TRUE if forwarding for busy and no answer can be use different destination addresses; FALSE otherwise.

LINEADDRCAPFLAGS_ACCEPTTOALERT
This flag is TRUE if an offering call must be accepted using **lineAccept** in order to start alerting the users at both ends of the call; Otherwise, it is FALSE. This flag is typically only used with ISDN.

LINEADDRCAPFLAGS_CONFDROP
This flag is TRUE if **lineDrop** of a conference call parent also has the side effect of dropping (disconnecting) the other parties involved in the conference call. It is FALSE if dropping a conference call still allows the other parties to talk among themselves.

LINEADDRCAPFLAGS_PICKUPCALLWAIT
This flag is TRUE if **linePickup** can be used to pick up a call detected by the user as a call waiting call. Otherwise, it is FALSE.

dwCallFeatures

This field specifies the switching capabilities or features available for all calls on this address, of type **LINECALLFEATURE**. Invoking a supported feature requires the call to be in the proper state and the underlying line device to be opened in a compatible mode. A zero in a bit position indicates that the corresponding feature is never available; a one indicates that the corresponding feature may be available if the application has the right privileges to the call, and the call is in the appropriate state for the operation to be meaningful. This field allow an aplication to discover early on which call features can be and which can never be supported by the address.

dwRemoveFromConfCaps

This field specifies the address's capabilities for removing calls from a conference call, and uses the following LINEREMOVEFROMCONF_ constants:

LINEREMOVEFROMCONF_NONE
Parties cannot be removed from the conference call.

LINEREMOVEFROMCONF_LAST
Only the most recently added party can be removed from the conference call.

LINEREMOVEFROMCONF_ANY
Any participating party can be removed from the conference call.

dwRemoveFromConfState

This field is of the type **LINECALLSTATE**, and it specifies the state of the call after it has been removed from a conference call.

dwTransferModes

This field specifies the address's capabilities for resolving transfer requests and uses the following LINETRANSFERMODE_ constants:

LINETRANSFERMODE_TRANSFER
Resolve the initiated transfer by transferring the initial call to the consultation call.

LINETRANSFERMODE_CONFERENCE
Resolve the initiated transfer by conferencing all three parties into a three-way conference call. A conference call is created and returned to the application.

dwParkModes

Specifies the different call park modes available at this address, and uses the following LINEPARKMODE_ constants:

LINEPARKMODE_DIRECTED
Specifies directed call park. The address where the call is to be parked must be supplied to the

switch.

LINEPARKMODE_NONDIRECTED
Specifies nondirected call park. The address where the call is parked is selected by the switch and provided by the switch to the application.

dwForwardModes

Specifies the different modes of forwarding available for this address and uses the following LINEFORWARDMODE_ constants:

LINEFORWARDMODE_UNCOND
Forward all calls unconditionally, irrespective of their origin. Use this value when unconditional forwarding for internal and external calls cannot be controlled separately. Unconditional forwarding overrides forwarding on busy and/or no answer conditions.

LINEFORWARDMODE_UNCONDINTERNAL
Forward all internal calls unconditionally. Use this value when unconditional forwarding for internal and external calls can be controlled separately.

LINEFORWARDMODE_UNCONDEXTERNAL
Forward all external calls unconditionally. Use this value when unconditional forwarding for internal and external calls can be controlled separately.

LINEFORWARDMODE_UNCONDSPECIFIC
Forward all calls that originated at a specified address unconditionally (selective call forwarding).

LINEFORWARDMODE_BUSY
Forward all calls on busy, irrespective of their origin. Use this value when forwarding for internal and external calls on busy and no answer cannot be controlled separately.

LINEFORWARDMODE_BUSYINTERNAL
Forward all internal calls on busy. Use this value when forwarding for internal and external calls on busy and no answer can be controlled separately.

LINEFORWARDMODE_BUSYEXTERNAL
Forward all external calls on busy. Use this value when forwarding for internal and external calls on busy and no answer can be controlled separately.

LINEFORWARDMODE_BUSYSPECIFIC
Forward all calls that originated at a specified address on busy (selective call forwarding).

LINEFORWARDMODE_NOANSW
Forward all calls on no answer, irrespective of their origin. Use this value when call forwarding for internal and external calls on no answer cannot be controlled separately.

LINEFORWARDMODE_NOANSWINTERNAL
Forward all internal calls on no answer. Use this value when forwarding for internal and external calls on no answer can be controlled separately.

LINEFORWARDMODE_NOANSWEXTERNAL
Forward all external calls on no answer. Use this value when forwarding for internal and external calls on no answer can be controlled separately.

LINEFORWARDMODE_NOANSWSPECIFIC
Forward all calls that originated at a specified address on no answer (selective call forwarding).

LINEFORWARDMODE_BUSYNA
Forward all calls on busy/no answer irrespective of their origin. Use this value when forwarding for internal and external calls on busy and no answer cannot be controlled separately.

LINEFORWARDMODE_BUSYNAINTERNAL
Forward all internal calls on busy/no answer. Use this value when call forwarding on busy and no answer cannot be controlled separately for internal calls.

LINEFORWARDMODE_BUSYNAEXTERNAL
Forward all external calls on busy/no answer. Use this value when call forwarding on busy and no answer cannot be controlled separately for internal calls.

LINEFORWARDMODE_BUSYNASPECIFIC
Forward all calls that originated at a specified address on busy/no answer (selective call forwarding).

dwMaxForwardEntries

Specifies the maximum number of entries that can be passed to **TSPI_lineForward** in the *lpForwardList* parameter.

dwMaxSpecificEntries

Specifies the maximum number of entries in the *lpForwardList* parameter passed to **TSPI_lineForward** that can contain forwarding instructions based on a specific caller ID (selective call forwarding). This field is zero if selective call forwarding is not supported.

dwMinFwdNumRings

Specifies the minimum number of rings that can be set to determine when a call is officially considered “no answer.”

dwMaxFwdNumRings

Specifies the maximum number of rings that can be set to determine when a call is officially considered “no answer.” If this number of rings cannot be set, then **dwMinFwdNumRings** and **dwMaxNumRings** will be equal.

dwMaxCallCompletions

Specifies the maximum number of concurrent call completion requests can be outstanding on this line device. Zero implies that call completion is not available.

dwCallCompletionCond

Specifies the different call conditions under which call completion can be requested. This parameter uses the following LINECALLCOMPLCOND_ constants:

LINECALLCOMPLCOND_BUSY
Complete the call under the busy condition.

LINECALLCOMPLCOND_NOANSWER
Complete the call under the ringback no answer condition.

dwCallCompletionModes

Specifies the way in which the call can be completed. This parameter uses the following LINECALLCOMPLMODE_ constants:

LINECALLCOMPLMODE_CAMPON
Queues the call until the call can be completed.

LINECALLCOMPLMODE_CALLBACK
Requests the called station to return the call when it returns to idle.

LINECALLCOMPLMODE_INTRUDE
Adds the application to the existing call at the called station if busy (barge in).

LINECALLCOMPLMODE_MESSAGE
Leave a short predefined message for the called station (Leave Word Calling). A specific message can be identified.

dwNumCompletionMessages

Specifies the number of call completion messages that can be selected from when using the LINECALLCOMPLMODE_MESSAGE option. Individual messages are identified by values in the range 0 through **dwNumCompletionMessages** minus one.

dwCompletionMsgTextEntrySize

Specifies the size in bytes of each of the call completion text descriptions pointed at by **dwCompletionMsgTextSize/Offset**.

dwCompletionMsgTextSize
dwCompletionMsgTextOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field containing descriptive text about each of the call completion messages. Each message is **dwCompletionMsgTextEntrySize** bytes long. The string format of these textual descriptions is indicated by **dwStringFormat** in the line’s device capabilities.

Device-specific extensions should use the DevSpecific (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

LINEADDRESSSTATUS

[New - Windows 95]

```
typedef struct lineaddressstatus_tag {
  DWORD  dwTotalSize;
  DWORD  dwNeededSize;
  DWORD  dwUsedSize;

  DWORD  dwNumInUse;
  DWORD  dwNumActiveCalls;
  DWORD  dwNumOnHoldCalls;
  DWORD  dwNumOnHoldPendCalls;
  DWORD  dwAddressFeatures;

  DWORD  dwNumRingsNoAnswer;
  DWORD  dwForwardNumEntries;
  DWORD  dwForwardSize;
  DWORD  dwForwardOffset;

  DWORD  dwTerminalModesSize;
  DWORD  dwTerminalModesOffset;

  DWORD  dwDevSpecificSize;
  DWORD  dwDevSpecificOffset;
} LINEADDRESSSTATUS, FAR *LPLINEADDRESSSTATUS;
```

The **LINEADDRESSSTATUS** structure describes the current status of an address.

dwTotalSize
The total size in bytes allocated to this data structure.

dwNeededSize
The size in bytes for this data structure that is needed to hold all the returned information.

dwUsedSize
The size in bytes of the portion of this data structure that contains useful information.

dwNumInUse
Specifies the number of stations that are currently using the address.

dwNumActiveCalls
The number of calls on the address that are in call states other than *idle*, *onHold*, ***onHoldPendingTransfer***, and ***OnHoldPendingConference***.

dwNumOnHoldCalls
The number of calls on the address in the *onHold* state.

dwNumOnHoldPendCalls
The number of calls on the address in the *onHoldPendingTransfer* or *onHoldPendingConference* state.

dwAddressFeatures
This field specifies the address-related API functions that can be invoked on the address in its current state. This field uses the following LINEADDRFEATURE_ constants:

LINEADDRFEATURE_FORWARD
The address can be forwarded.

LINEADDRFEATURE_MAKECALL
An outbound call can placed on the address.

LINEADDRFEATURE_PICKUP
A call can be picked up at the address.

LINEADDRFEATURE_SETMEDIACONTROL
Media control can be set on this address.

LINEADDRFEATURE_SETTERMINAL

The terminal modes for this address can be set.

LINEADDRFEATURE_SETUPCONF
A conference call with a NULL initial call can be set up at this address.

LINEADDRFEATURE_UNCOMPLETECALL
Call completion requests can be canceled at this address.

LINEADDRFEATURE_UNPARK
Calls can be unparked using this address.

dwNumRingsNoAnswer
Specifies the number of rings set for this address before an unanswered call is considered as no answer.

dwForwardNumEntries
Specifies the number of entries in the array referred to by **dwForwardSize** and **dwForwardOffset**.

dwForwardSize
dwForwardOffset
The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized field that describes the address's forwarding information. This information is an array of **dwForwardNumEntries** elements, of type **LINEFORWARD**. The offsets of the addresses in the array are relative to the beginning of the **LINEADDRESSTATUS** structure.

dwTerminalModesSize
dwTerminalModesOffset
The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device field containing an array with **DWORD**-sized entries, of type **LINETERMMODE**. This array is indexed by terminal IDs, in the range from zero to **dwNumTerminals** minus one. Each entry in the array specifies the current terminal modes for the corresponding terminal set through the **TSPI_lineSetTerminal** operation for this address. Values are:

LINETERMMODE_BUTTONS
These are button press events sent from the terminal to the line.

LINETERMMODE_LAMPS
This are lamp events sent from the line to the terminal.

LINETERMMODE_DISPLAY
This is display information sent from the line to the terminal.

LINETERMMODE_RINGER
This is ringer control information sent from the switch to the terminal.

LINETERMMODE_HOOKSWITCH
These are hookswitch event sent between the terminal and the line.

LINETERMMODE_MEDIATOLINE
This is the unidirectional media stream from the terminal to the line associated with a call on the line. Use this value when routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIAFROMLINE
This is the unidirectional media stream from the line to the terminal associated with a call on the line. Use this value when routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIABIDIRECT
This is the bidirectional media stream associated with a call on the line and the terminal. Use this value when routing of both unidirectional channels of a call's media stream cannot be controlled independently.

dwDevSpecificSize
dwDevSpecificOffset
The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device-specific field.

Device-specific extensions should use the DevSpecific (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

This data structure is returned by **TSPI_lineGetAddressStatus**. When items in this data structure change as a consequence of activities on the address, a **LINE_ADDRESSSTATE** message is sent to

the application. A parameter to this message is the address state, of type LINEADDRESSSTATE_, which is an an indication of the status item in this record that changed.

LINECALLINFO

[New - Windows 95]

```
typedef struct linecallinfo_tag {
  DWORD  dwTotalSize;
  DWORD  dwNeededSize;
  DWORD  dwUsedSize;

  HLINE  hLine;
  DWORD  dwLineDeviceID;
  DWORD  dwAddressID;

  DWORD  dwBearerMode;
  DWORD  dwRate;
  DWORD  dwMediaMode;

  DWORD  dwAppSpecific;
  DWORD  dwCallID;
  DWORD  dwRelatedCallID;
  DWORD  dwCallParamFlags;
  DWORD  dwCallStates;

  DWORD  dwMonitorDigitModes;
  DWORD  dwMonitorMediaModes;
  LINEDIALPARAMS  DialParams;

  DWORD  dwOrigin;
  DWORD  dwReason;
  DWORD  dwCompletionID;
  DWORD  dwNumOwners;
  DWORD  dwNumMonitors;

  DWORD  dwCountryCode;
  DWORD  dwTrunk;

  DWORD  dwCallerIDFlags;
  DWORD  dwCallerIDSize;
  DWORD  dwCallerIDOffset;
  DWORD  dwCallerIDNameSize;
  DWORD  dwCallerIDNameOffset;

  DWORD  dwCalledIDFlags;
  DWORD  dwCalledIDSize;
  DWORD  dwCalledIDOffset;
  DWORD  dwCalledIDNameSize;
  DWORD  dwCalledIDNameOffset;

  DWORD  dwConnectedIDFlags;
  DWORD  dwConnectedIDSize;
  DWORD  dwConnectedIDOffset;
  DWORD  dwConnectedIDNameSize;
  DWORD  dwConnectedIDNameOffset;

  DWORD  dwRedirectionIDFlags;
  DWORD  dwRedirectionIDSize;
  DWORD  dwRedirectionIDOffset;
  DWORD  dwRedirectionIDNameSize;
  DWORD  dwRedirectionIDNameOffset;

  DWORD  dwRedirectingIDFlags;
  DWORD  dwRedirectingIDSize;
  DWORD  dwRedirectingIDOffset;
  DWORD  dwRedirectingIDNameSize;
```

```
  DWORD  dwRedirectingIDNameOffset;

  DWORD  dwAppNameSize;
  DWORD  dwAppNameOffset;

  DWORD  dwDisplayableAddressSize
  DWORD  dwDisplayableAddressOffset

  DWORD  dwCalledPartySize;
  DWORD  dwCalledPartyOffset;

  DWORD  dwCommentSize;
  DWORD  dwCommentOffset;

  DWORD  dwDisplaySize;
  DWORD  dwDisplayOffset;

  DWORD  dwUserUserInfoSize;
  DWORD  dwUserUserInfoOffset;

  DWORD  dwHighLevelCompSize;
  DWORD  dwHighLevelCompOffset;

  DWORD  dwLowLevelCompSize;
  DWORD  dwLowLevelCompOffset;

  DWORD  dwChargingInfoSize;
  DWORD  dwChargingInfoOffset;

  DWORD  dwTerminalModesSize;
  DWORD  dwTerminalModesOffset;

  DWORD  dwDevSpecificSize;
  DWORD  dwDevSpecificOffset;
} LINECALLINFO, FAR *LPLINECALLINFO;
```

The **LINECALLINFO** structure contains information about a call. This information remains relatively fixed for the duration of the call and is obtained using **TSPI_lineGetCallInfo**. If a part of the structure does change, a **LINE_CALLINFO** message is sent to the application indicating which information item has changed. Dynamically changing information about a call, such as call progress status, is available in the **LINECALLSTATUS** structure, returned with the function **TSPI_lineGetCallStatus**.

dwTotalSize
The total size in bytes allocated to this data structure.

dwNeededSize
The size in bytes for this data structure that is needed to hold all the returned information.

dwUsedSize
The size in bytes of the portion of this data structure that contains useful information.

hLine
Specifies the handle for the line device with which this call is associated.

dwLineDeviceID
Specifies the device ID of the line device with which this call is associated.

dwAddressID
Specifies the address ID of the address on the line on which this call exists.

dwBearerMode
Specifies the current bearer mode of the call. This field uses the following LINEBEARERMODE_ constants:

LINEBEARERMODE_VOICE
This is a regular 3.1kHz analog voice grade bearer service. Bit integrity is not assured. Voice can support fax and modem media modes.

LINEBEARERMODE_SPEECH
This corresponds to G.711 speech transmission on the call. The network may use processing techniques such as analog transmission, echo cancellation and compression/decompression. Bit integrity is not assured. Speech is not intended to support fax and modem media modes.

LINEBEARERMODE_MULTIUSE
The multi-use mode defined by ISDN.

LINEBEARERMODE_DATA
The unrestricted data transfer on the call. The data rate is specified separately.

LINEBEARERMODE_ALTSPEECHDATA
The alternate transfer of speech or unrestricted data on the same call (ISDN).

LINEBEARERMODE_NONCALLSIGNALING
This corresponds to a non-call-associated signaling connection from the application to the service provider or switch (treated as a “media stream” by the Telephony API).

dwRate

Specifies the rate of the call’s data stream in bps (bits per second).

dwMediaMode

Specifies the media mode of the information stream currently on the call, of type LINEMEDIAMODE_. This is the media mode as determined by the owner of the call, which is not necessarily the same as that of the last **LINE_MONITORMEDIA** message. This field is not directly affected by the **LINE_MONITORMEDIA** messages. Values are:

LINEMEDIAMODE_UNKNOWN
A media stream exists but its mode is not known. This corresponds to a call with an unclassified media type. In typical analog telephony environments, an inbound call’s media mode may be unknown until after the call has been answered and the media stream filtered to make a determination.

LINEMEDIAMODE_INTERACTIVEVOICE
The presence of voice energy on the call and the call is treated as an interactive call with humans on both ends.

LINEMEDIAMODE_AUTOMATEDVOICE
The presence of voice energy on the call and the voice is locally handled by an automated application.

LINEMEDIAMODE_DATAMODEM
A data modem session on the call.

LINEMEDIAMODE_G3FAX
A group 3 fax is being sent or received over the call.

LINEMEDIAMODE_TDD
A TDD (Telephony Devices for the Deaf) session on the call.

LINEMEDIAMODE_G4FAX
A group 4 fax is being sent or received over the call.

LINEMEDIAMODE_DIGITALDATA
Digital data being sent or received over the call.

LINEMEDIAMODE_TELETEX
A teletex session on the call. Teletex is one of the telematic services.

LINEMEDIAMODE_VIDEOTEX
A videotex session on the call. Videotex is one the telematic services.

LINEMEDIAMODE_TELEX
A telex session on the call. Telex is one the telematic services.

LINEMEDIAMODE_MIXED
A mixed session on the call. Mixed is one the ISDN telematic services.

LINEMEDIAMODE_ADSI
An ADSI (Analog Display Services Interface) session on the call.

dwAppSpecific

This field is uninterpreted by the API implementation and service provider. It can be set by any

owner application of this call with **TSPI_lineSetAppSpecific**.

dwCallID

In some telephony environments, the switch or service provider may assign a unique identifier to each call. This allows the call to be tracked across transfers, forwards, or other events. The domain of these call IDs and their scope is service provider defined. The **dwCallID** field makes this unique identifier available to the applications.

dwRelatedCallID

Telephony environments that use the call ID often may find it necessary to relate one call to another. The **dwRelatedCallID** field may be used by the service provider for this purpose.

dwCallParamFlags

Specifies a collection of call-related parameters when the call is outbound. These are the same call parameters specified through **TSPI_lineMakeCall**, of type LINECALLPARAMFLAGS_. Values are:

LINECALLPARAMFLAGS_SECURE
The call is currently secure. This flag is also updated if the call is later secured with **TSPI_lineSecureCall**.

LINECALLPARAMFLAGS_IDLE
The call started out using an idle call.

LINECALLPARAMFLAGS_BLOCKID
The originator identity was concealed (block caller ID presentation to the remote party).

LINECALLPARAMFLAGS_ORIGOFFHOOK
The originator's phone was automatically taken off hook.

LINECALLPARAMFLAGS_DESTOFFHOOK
The called party's phone was automatically taken off hook.

dwCallStates

Specifies the call states for which the application may be notified on this call. This field uses the LINECALLSTATE_ constants listed below. The **dwCallStates** field is constant in **LINECALLINFO**. It describes the call states that can be reported during the lifetime of this call, and does not change depending on call state. (**LINECALLINFO** should be seen as a "call caps".) Values are:

LINECALLSTATE_IDLE
The call is idle—no call actually exists.

LINECALLSTATE_OFFERING
The call is being offered to the station, signaling the arrival of a new call. In some environments, a call in the *offering* state does not automatically alert the user; alerting is done by the switch instructing the line to ring, it does not affect any call states.

LINECALLSTATE_ACCEPTED
The call was offering and has been accepted. This indicates to other (monitoring) applications that the current owner application has claimed responsibility for answering the call. In ISDN, this also initiates alerting to both parties.

LINECALLSTATE_DIALTONE
The call is receiving a dial tone from the switch, which means that the switch is ready to receive a dialed number.

LINECALLSTATE_DIALING
Destination address information (a phone number) is being sent to the switch over the call. Note that the operation **TSPI_lineGenerateDigits** does not place the line into the *dialing* state.

LINECALLSTATE_RINGBACK
The call is receiving ringback from the called address. Ringback indicates that the other station has been reached and is being alerted.

LINECALLSTATE_BUSY
The call is receiving a busy tone. Busy tone indicates that the call cannot be completed because either a circuit (trunk) or the remote party's station are in use.

LINECALLSTATE_SPECIALINFO
Special information is sent by the network. Special information is typically sent when the destination cannot be reached.

LINECALLSTATE_CONNECTED
The call has been established, the connection is made. Information is able to flow over the call between the originating address and the destination address.

LINECALLSTATE_PROCEEDING
Dialing has completed and the call is proceeding through the switch or telephone network.
LINECALLSTATE_ONHOLD
The call is on hold by the switch.
LINECALLSTATE_CONFERENCED
The call is currently a member of a multi-party conference call.
LINECALLSTATE_ONHOLDPENDCONF
The call is currently on hold while it is being added to a conference.
LINECALLSTATE_ONHOLDPENDTRANSFER
The call is currently on hold awaiting transfer to another number.
LINECALLSTATE_DISCONNECTED
The remote party has disconnected from the call.
LINECALLSTATE_UNKNOWN
The state of the call is not known. This may be due to limitations of the call progress detection implementation.

dwMonitorDigitsModes

Specifies the various digit modes for which monitoring is currently enabled. This field uses the following LINEDIGITMODE_ constants:

LINEDIGITMODE_PULSE
Uses pulse/rotary for digit signaling.
LINEDIGITMODE_DTMF
Uses DTMF tones for digit signaling.
LINEDIGITMODE_DTMFEND
Uses DTMF tones for digit detection, and also detects the down edges.

dwMonitorMediaModes

Specifies the various media modes for which monitoring is currently enabled. This field uses the following LINEMEDIAMODE_ constants:

LINEMEDIAMODE_INTERACTIVEVOICE
The presence of voice energy on the call and the call is treated as an interactive call with humans on both ends.
LINEMEDIAMODE_AUTOMATEDVOICE
The presence of voice energy on the call and the voice is locally handled by an automated application.
LINEMEDIAMODE_DATAMODEM
A data modem session on the call.
LINEMEDIAMODE_G3FAX
A group 3 fax is being sent or received over the call.
LINEMEDIAMODE_TDD
A TDD (Telephony Devices for the Deaf) session on the call.
LINEMEDIAMODE_G4FAX
A group 4 fax is being sent or received over the call.
LINEMEDIAMODE_DIGITALDATA
Digital data is being sent or received over the call.
LINEMEDIAMODE_TELETEX
A teletex session on the call. Teletex is one of the telematic services.
LINEMEDIAMODE_VIDEOTEX
A videotex session on the call. Videotex is one the telematic services.
LINEMEDIAMODE_TELEX
A telex session on the call. Telex is one the telematic services.
LINEMEDIAMODE_MIXED
A mixed session on the call. Mixed is one the ISDN telematic services.
LINEMEDIAMODE_ADSI
An ADSI (Analog Display Services Interface) session on the call.

DialParams

Specifies the dialing parameters currently in effect on the call, of type **LINEDIALPARAMS**. Unless

these parameters are set by either **TSPI_lineMakeCall** or **TSPI_lineSetCallParams**, their values will be the same as the defaults used in the **LINEDEVCAPS**.

dwOrigin

Identifies where the call originated from. This field uses the following LINECALLORIGIN_ constants:

LINECALLORIGIN_OUTBOUND
The call is an outbound call.

LINECALLORIGIN_INTERNAL
The call is inbound and originated internally (on the same PBX, for example).

LINECALLORIGIN_EXTERNAL
The call is inbound and originated externally.

LINECALLORIGIN_UNKNOWN
The call is an inbound call, but its origin is currently unknown but may become known later.

LINECALLORIGIN_UNAVAIL
The call is an inbound call, and its origin is not available and will never become known for this call.

LINECALLORIGIN_CONFERENCE
The call handle is for a conference call, that is, the application's connection to the conference bridge in the switch.

dwReason

Specifies the reason why the call occurred. This field uses the following LINECALLREASON_ constants:

LINECALLREASON_DIRECT
This is a direct call.

LINECALLREASON_FWDBUSY
This call was forwarded from another extension that was busy at the time of the call.

LINECALLREASON_FWDNOANSWER
The call was forwarded from another extension that didn't answer the call after some number of rings.

LINECALLREASON_FWDUNCOND
The call was forwarded unconditionally from another number.

LINECALLREASON_PICKUP
The call was picked up from another extension.

LINECALLREASON_UNPARK
The call was retrieved as a parked call.

LINECALLREASON_REDIRECT
The call was redirected to this station.

LINECALLREASON_CALLCOMPLETION
The call was the result of a call completion request.

LINECALLREASON_TRANSFER
The call has been transferred from another number. Party ID information may indicate who the caller is and where the call was transferred from.

LINECALLREASON_REMINDER
The call is a reminder (or "recall") that the user has a call parked or on hold for potentially a long time.

LINECALLREASON_UNKNOWN
The reason for the call is currently unknown but may become known later.

LINECALLREASON_UNAVAIL
The reason for the call is unavailable and will not become known later.

dwCompletionID

The completion ID for the incoming call if it is the result of a completion request that terminates. This ID is meaningful only if **dwReason** is LINECALLREASON_CALLCOMPLETION.

dwNumOwners

The number of application modules with different call handles with owner privilege for the call.

dwNumMonitors

The number of application modules with different call handles with monitor privilege for the call.

dwCountryCode

The country code of the destination party. Zero if unknown.

dwTrunk

The number of the trunk over which the call is routed. This field is used for both inbound and outgoing calls. **dwTrunk** should be set to 0xFFFFFFFF if it is unknown.

dwCallerIDFlags

Determines the validity and content of the caller party ID information. The caller is the originator of the call. This field uses the following LINECALLPARTYID_ constants:

LINECALLPARTYID_BLOCKED
: Caller ID information for the call has been blocked by the caller, but would otherwise have been available.

LINECALLPARTYID_OUTOFAREA
: Caller ID information for the call is not available since it is not propagated all the way by the network.

LINECALLPARTYID_NAME
: The caller ID information for the call is the caller's name (from a table maintained inside the switch). It is provided in the caller ID name variably sized field.

LINECALLPARTYID_ADDRESS
: The caller ID information for the call is the caller's number, and is provided in the caller ID variably sized field.

LINECALLPARTYID_PARTIAL
: Caller ID information for the call is valid, but is limited to partial number information.

LINECALLPARTYID_UNKNOWN
: Caller ID information is currently unknown, but it may become known later.

LINECALLPARTYID_UNAVAIL
: Caller ID information is unavailable and will not become known later.

dwCallerIDSize
dwCallerIDOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized field containing the caller party ID number information.

dwCallerIDNameSize
dwCallerIDNameOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized field containing the caller party ID name information.

dwCalledIDFlags

Determines the validity and content of the called party ID information. The called party corresponds to the orignally addressed party. This field uses the following LINECALLPARTYID_ constants:

LINECALLPARTYID_BLOCKED
: Called ID information for the call has been blocked by the caller, but would otherwise have been available.

LINECALLPARTYID_OUTOFAREA
: Caller ID information for the call is not available as it is not propagated all the way by the network.

LINECALLPARTYID_NAME
: The called ID information for the call is the caller's name (from a table maintained inside the switch). It is provided in the called ID name variably sized field.

LINECALLPARTYID_ADDRESS
: The called ID information for the call is the caller's number, and is provided in the called ID variably sized field.

LINECALLPARTYID_PARTIAL
: Called ID information for the call is valid, but is limited to partial number information.

LINECALLPARTYID_UNKNOWN
: Called ID information is currently unknown but it may become known later.

LINECALLPARTYID_UNAVAIL
: Called ID information is unavailable and will not become known later.

dwCalledIDSize

dwCalledIDOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized field containing the called party ID number information.

dwCalledIDNameSize
dwCalledIDNameOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized field containing the called party ID name information.

dwConnectedFlags

Determines the validity and content of the connected party ID information. The connected party is the party that was actually connected to. This may be different from the called party ID if the call was diverted. This field uses the following LINECALLPARTYID_ constants:

LINECALLPARTYID_BLOCKED
Connected party ID information for the call has been blocked by the caller, but would otherwise have been available.

LINECALLPARTYID_OUTOFAREA
Connected ID information for the call is not available as it is not propagated all the way by the network.

LINECALLPARTYID_NAME
The connected party ID information for the call is the caller's name (from a table maintained inside the switch). It is provided in the connected ID name variably sized field.

LINECALLPARTYID_ADDRESS
The connected party ID information for the call is the caller's number, and is provided in the connected ID variably sized field.

LINECALLPARTYID_PARTIAL
Connected party ID information for the call is valid, but is limited to partial number information.

LINECALLPARTYID_UNKNOWN
Connected party ID information is currently unknown, it may become known later.

LINECALLPARTYID_UNAVAIL
Connected party ID information is unavailable and will not become known later.

dwConnectedIDSize
dwConnectedIDOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized field containing the connected party ID number information.

dwConnectedIDNameSize
dwConnectedIDNameOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized field containing the connected party ID name information.

dwRedirectionIDFlags

Determines the validity and content of the redirection party ID information. The redirection party identifies to the calling user the number towards which diversion was invoked. This field uses the following LINECALLPARTYID_ constants:

LINECALLPARTYID_BLOCKED
Redirection party ID information for the call has been blocked by the caller, but would otherwise have been available.

LINECALLPARTYID_OUTOFAREA
Redirection ID information for the call is not available as it is not propagated all the way by the network.

LINECALLPARTYID_NAME
The redirection party ID information for the call is the caller's name (from a table maintained inside the switch). It is provided in the redirection ID name variably sized field.

LINECALLPARTYID_ADDRESS
The redirection party ID information for the call is the caller's number, and is provided in the redirection ID variably sized field.

LINECALLPARTYID_PARTIAL
Redirection party ID information for the call is valid, but is limited to partial number information.

LINECALLPARTYID_UNKNOWN
Redirection ID information is currently unknown but it may become known later.
LINECALLPARTYID_UNAVAIL
Redirection ID information is unavailable and will not become known later.

dwRedirectionIDSize
dwRedirectionIDOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized field containing the redirection party ID number information.

dwRedirectionIDNameSize
dwRedirectionIDNameOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized field containing the redirection party ID name information.

dwRedirectingIDFlags

Determines the validity and content of the redirecting party ID information. The redirecting party identifies to the diverted-to user the party from which diversion was invoked. This field uses the following LINECALLPARTYID_ constants:

LINECALLPARTYID_BLOCKED
Redirecting party ID information for the call has been blocked by the caller, but would otherwise have been available.
LINECALLPARTYID_OUTOFAREA
Redirecting ID information for the call is not available since it is not propagated all the way by the network.
LINECALLPARTYID_NAME
The redirecting party ID information for the call is the caller's name (from a table maintained inside the switch). It is provided in the redirecting ID name variably sized field.
LINECALLPARTYID_ADDRESS
The redirecting party ID information for the call is the caller's number, and is provided in the redirecting ID variably sized field.
LINECALLPARTYID_PARTIAL
Redirecting party ID information for the call is valid, but is limited to partial number information.
LINECALLPARTYID_UNKNOWN
Redirecting ID information is currently unknown, it may become known later.
LINECALLPARTYID_UNAVAIL
Redirecting ID information is unavailable and will not become known later.

dwRedirectingIDSize
dwRedirectingIDOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized field containing the redirecting party ID number information.

dwRedirectingIDNameSize
dwRedirectingIDNameOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized field containing the redirecting party ID name information.

dwAppNameSize
dwAppNameOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field holding the user-friendly application name of the application that first originated, accepted, or answered the call. This is the name that an application can specify with **lineInitialize**. If the application specifies no such name, the application's module name is used instead.

dwDisplayableAddressSize
dwDisplayableAddressOffset

The displayable string is used for logging purposes. The information is obtained from **LINECALLPARAMS** for functions that intiate calls. The TAPI function **lineTranslateAddress** returns appropriate information to be placed in this field in the **dwDisplayableAddressSize** and **dwDisplayableAddressOffset** fields of the **LINETRANSLATEOUTPUT** structure.

dwCalledPartySize

dwCalledPartyOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field holding a user-friendly description of the called party. This information can be specified with **TSPI_lineMakeCall** and can be optionally specified with the *lpCallParams* whenever a new call is established. It is useful for call logging purposes.

dwCommentSize
dwCommentOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field holding a comment about the call provided by the application that originated the call using **TSPI_lineMakeCall**. This information can be optionally specified with the *lpCallParams* whenever a new call is established.

dwDisplaySize
dwDisplayOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field holding raw display information. Depending on the telephony environment, a service provider may extract functional information from this for presentation into a more functional way.

dwUserUserInfoSize
dwUserUserInfoOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field holding user-to-user information.

dwHighLevelCompSize
dwHighLevelCompOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field holding high level compatibility information. The format of this information is specified by other standards (ISDN Q.931).

dwLowLevelCompSize
dwLowLevelCompOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field holding low level compatibility information. The format of this information is specified by other standards (ISDN Q.931).

dwChargingInfoSize
dwChargingInfoOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field holding charging information. The format of this information is specified by other standards (ISDN Q.931).

dwTerminalModesSize
dwTerminalModesOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device field containing an array with DWORD-sized entries, of type LINETERMMODE_. This array is indexed by terminal IDs, in the range from zero to **dwNumTerminals** minus one. Each entry in the array specifies the current terminal modes for the corresponding terminal set with the **TSPI_lineSetTerminal** operation for this call's media stream. Values are:

LINETERMMODE_BUTTONS
: These are button press events sent from the terminal to the line.

LINETERMMODE_LAMPS
: This are lamp events sent from the line to the terminal.

LINETERMMODE_DISPLAY
: This is display information sent from the line to the terminal.

LINETERMMODE_RINGER
: This is ringer control information sent from the switch to the terminal.

LINETERMMODE_HOOKSWITCH
: These are hookswitch event sent between the terminal and the line.

LINETERMMODE_MEDIATOLINE
: This is the unidirectional media stream from the terminal to the line associated with a call on the line. Use this value when routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIAFROMLINE
This is the unidirectional media stream from the line to the terminal associated with a call on the line. Use this value when routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIABIDIRECT
This is the bidirectional media stream associated with a call on the line and the terminal. Use this value when routing of both unidirectional channels of a call's media stream cannot be controlled independently.

dwDevSpecificSize
dwDevSpecificOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field holding device-specific information.

Device-specific extensions should use the DevSpecific (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

The **LINECALLINFO** data structure contains relatively fixed information about a call. This structure is returned with **TSPI_lineGetCallInfo**. When information items in this data structure have changed, a **LINE_CALLINFO** message is sent to the application. A parameter to this message is the information item or field that changed.

The **dwAppSpecific** field can be used by applications to tag calls by using **TSPI_lineSetAppSpecific**. This field is uninterpreted by TAPI or service providers. It is initially set to zero.

LINECALLPARAMS

[New - Windows 95]

```
typedef struct linecallparams_tag {  // Defaults:
  DWORD  dwTotalSize;  // ---------

  DWORD  dwBearerMode;  // voice
  DWORD  dwMinRate;  // (3.1kHz)
  DWORD  dwMaxRate;  // (3.1kHz)
  DWORD  dwMediaMode;  // interactiveVoice

  DWORD  dwCallParamFlags;  // 0
  DWORD  dwAddressMode;  // addressID
  DWORD  dwAddressID  // (any available)

  LINEDIALPARAMS DialParams;  // (0, 0, 0, 0)

  DWORD  dwOrigAddressSize;  // 0
  DWORD  dwOrigAddressOffset;

  DWORD  dwDisplayableAddressSize;  // 0
  DWORD  dwDisplayableAddressOffset;

  DWORD  dwCalledPartySize;  // 0
  DWORD  dwCalledPartyOffset;

  DWORD  dwCommentSize;  // 0
  DWORD  dwCommentOffset;

  DWORD  dwUserUserInfoSize;  // 0
  DWORD  dwUserUserInfoOffset;

  DWORD  dwHighLevelCompSize;  // 0
  DWORD  dwHighLevelCompOffset;

  DWORD  dwLowLevelCompSize;  // 0
  DWORD  dwLowLevelCompOffset;

  DWORD  dwDevSpecificSize;  // 0
  DWORD  dwDevSpecificOffset;
} LINECALLPARAMS, FAR *LPLINECALLPARAMS;
```

The **LINECALLPARAMS** structure describes parameters supplied when making calls with **TSPI_lineMakeCall**. The **LINECALLPARAMS** structure is also used as a parameter in other operations. The comments on the right indicate the default values used when this structure is not provided to **TSPI_lineMakeCall**.

dwTotalSize

The total size in bytes allocated to this data structure. This size should be big enough to hold all the fixed and variably sized portions of this data structure.

dwBearerMode

Specifies the bearer mode for the call. This field uses the following LINEBEARERMODE_ constants:

LINEBEARERMODE_VOICE

This is a regular 3.1kHz analog voice grade bearer service. Bit integrity is not assured. Voice can support fax and modem media modes.

LINEBEARERMODE_SPEECH

This corresponds to G.711 speech transmission on the call. The network may use processing techniques such as analog transmission, echo cancellation and compression/decompression. Bit integrity is not assured. Speech is not intended to support fax and modem media modes.

LINEBEARERMODE_MULTIUSE
The multi-use mode defined by ISDN.

LINEBEARERMODE_DATA
The unrestricted data transfer on the call. The data rate is specified separately.

LINEBEARERMODE_ALTSPEECHDATA
The alternate transfer of speech or unrestricted data on the same call (ISDN).

LINEBEARERMODE_NONCALLSIGNALING
This corresponds to a non-call-associated signaling connection from the application to the service provider or switch (treated as a “media stream” by the Telephony API).

dwMinRate
dwMaxRate

Specifies the data rate range requested for the call’s data stream in bps (bits per second). When making a call, the service provider will attempt to provide the highest available rate in the requested range. If a specific data rate is required, both min and max should be set to that value. If an application works best with one rate, but is able the degrade to lower rates, the application should specify these as the max and min rates respectively.

dwMediaMode

Specifies the expected media mode of the call. This field uses the following LINEMEDIAMODE_ constants:

LINEMEDIAMODE_UNKNOWN
A media stream exists but its mode is not known. This corresponds to a call with an unclassified media type. In typical analog telephony environments, an inbound call’s media mode may be unknown until after the call has been answered and the media stream filtered to make a determination.

LINEMEDIAMODE_INTERACTIVEVOICE
The presence of voice energy on the call and the call is treated as an interactive call with humans on both ends.

LINEMEDIAMODE_AUTOMATEDVOICE
The presence of voice energy on the call and the voice is locally handled by an automated application.

LINEMEDIAMODE_DATAMODEM
A data modem session on the call.

LINEMEDIAMODE_G3FAX
A group 3 fax is being sent or received over the call.

LINEMEDIAMODE_TDD
A TDD (Telephony Devices for the Deaf) session on the call.

LINEMEDIAMODE_G4FAX
A group 4 fax is being sent or received over the call.

LINEMEDIAMODE_DIGITALDATA
Digital data being sent or received over the call.

LINEMEDIAMODE_TELETEX
A teletex session on the call. Teletex is one of the telematic services.

LINEMEDIAMODE_VIDEOTEX
A videotex session on the call. Videotex is one the telematic services.

LINEMEDIAMODE_TELEX
A telex session on the call. Telex is one the telematic services.

LINEMEDIAMODE_MIXED
A mixed session on the call. Mixed is one the ISDN telematic services.

LINEMEDIAMODE_ADSI
An ADSI (Analog Display Services Interface) session on the call.

dwCallParamFlags

These flags specify a collection of boolean call setup parameters, of type LINECALLPARAMFLAGS_. Values are:

LINECALLPARAMFLAGS_SECURE
The call should be set up as secure.

LINECALLPARAMFLAGS_IDLE

The call should get an idle call appearance.

LINECALLPARAMFLAGS_BLOCKID
The originator identity should be concealed (block caller ID).

LINECALLPARAMFLAGS_ORIGOFFHOOK
The originator's phone should be automatically taken off hook.

LINECALLPARAMFLAGS_DESTOFFHOOK
The called party's phone should be automatically be taken offhook.

dwAddressMode

Specifies the mode by which the originating address is specified, of type LINEADDRESSMODE_. Values are:

LINEADDRESSMODE_ADDRESSID
The address is specified by means of a small integer in the range 0 to *dwNumAddresses* minus one, where *dwNumAddresses* is the value in the line's device capabilities. The selected address is specified in the **dwAddressID** field.

LINEADDRESSMODE_DIALABLEADDR
The address is specified with its dialable address. Th eaddress is contained in the **dwOrigAddressSize**, **dwOrigAddressOffset** variably sized field.

dwAddressID

Specifies the address ID of the originating address if **dwAddressID** set to LINEADDRESSMODE_ADDRESSID.

DialParams

Specifies dial parameters to be used on this call, of type **LINEDIALPARAMS**. When a value of zero is specified for this field, the default value for the field is used. If a non-zero value is specified for the field which is outside the range specified by the corresponding fields in **MinDialParams** and **MaxDialParams** in the **LINEDEVCAPS** structure, the nearest value within the valid range is used instead.

dwOrigAddressSize
dwOrigAddressOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field holding the originating address. The format of this address depends on the **dwAddressMode** field.

dwDisplayableAddressSize
dwDisplayableAddressOffset

The displayable string is used for logging purposes. The content of these fields is recorded in the **dwDisplayableAddressSize/Offset** field of the call's **LINECALLINFO** message. This information is intended for logging purposes. The TAPI function **lineTranslateAddress** returns appropriate information to be placed in this field in the **dwDisplayableAddressSize/Offset** fields of the **LINETRANSLATEOUTPUT** structure.

dwCalledPartySize
dwCalledPartyOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field holding called party information. This information can be specified by the application that makes the call and is made available in the call's information structure for logging purposes. The format of this field is that of **dwStringFormat**, as specified in **LINEDEVCAPS**.

dwCommentSize
dwCommentOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field holding comments about the call. This information can be specified by the application the makes the call and is made available in the call's information structure for logging purposes. The format of this field is that of **dwStringFormat**, as specified in **LINEDEVCAPS**.

dwUserUserInfoSize
dwUserUserInfoOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field holding user-to-user information.

dwHighLevelCompSize

dwHighLevelCompOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field holding high level compatibility information.

dwLowLevelCompSize
dwLowLevelCompOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field holding low level compatibility information.

dwDevSpecificSize
dwDevSpecificOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field holding device-specific information.

Device-specific extensions should use the DevSpecific (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

This structure is used as a parameter for other operations, such as **TSPI_lineMakeCall** when setting up a call. Its fields allow the application to specify the quality of service requested from the network as well as a variety of ISDN call setup parameters. If no **LINECALLPARAMS** structure is supplied to **TSPI_lineMakeCall**, a default POTS voice grade call is requested with the default values listed above.

Note that the parameters **DialParams** through **dwDevSpecificOffset** are ignored when an *lpCallParams* parameter is specified with the function **TSPI_lineOpen**.

LINECALLSTATUS

[New - Windows 95]

```
typedef struct linecallstatus_tag {
  DWORD  dwTotalSize;
  DWORD  dwNeededSize;
  DWORD  dwUsedSize;

  DWORD  dwCallState;
  DWORD  dwCallStateMode;
  DWORD  dwCallPrivilege;
  DWORD  dwCallFeatures;

  DWORD  dwDevSpecificSize;
  DWORD  dwDevSpecificOffset;
} LINECALLSTATUS, FAR *LPLINECALLSTATUS;
```

The **LINECALLSTATUS** structure describes the current status of a call. The information in this structure, as returned with **TSPI_lineGetCallStatus**, depends on the device capabilities of the address, the ownership of the call by the invoking application, and the current state of the call being queried.

dwTotalSize

The total size in bytes allocated to this data structure.

dwNeededSize

The size in bytes for this data structure that is needed to hold all the returned information.

dwUsedSize

The size in bytes of the portion of this data structure that contains useful information.

dwCallState
dwCallStateMode

The **dwCallState** field specifies the current call state of the call, of type LINECALLSTATE_. The interpretation of the **dwCallStateMode** field is call-state-dependent. It specifies the current mode of the call while in its current state (if that state defines a mode).

LINECALLSTATE_IDLE
Call state mode is unused.

LINECALLSTATE_OFFERING
Call state mode is unused.

LINECALLSTATE_ACCEPTED
Call state mode is unused.

LINECALLSTATE_DIALTONE
The call state mode is of type LINEDIALTONEMODE_. Values are:

LINEDIALTONEMODE_NORMAL
This is a “normal” dialtone which typically is a continuous tone.

LINEDIALTONEMODE_SPECIAL
This is a special dialtone indicating a certain condition is currently in effect.

LINEDIALTONEMODE_INTERNAL
This an internal dialtone, as within a PBX.

LINEDIALTONEMODE_EXTERNAL
This is an external (public network) dialtone.

LINEDIALTONEMODE_UNKNOWN
The dialtone mode is currently known, but may become known later.

LINEDIALTONEMODE_UNAVAIL
The dialtone mode is unavailable and will not become known.

LINECALLSTATE_DIALING
Call state mode is unused.

LINECALLSTATE_RINGBACK
Call state mode is unused.

LINECALLSTATE_BUSY

The call state mode is of type LINEBUSYMODE_. Values are:

LINEBUSYMODE_STATION
: The busy signal indicates that the called party's station is busy. This is usually signaled by means of a "normal" busy tone.

LINEBUSYMODE_TRUNK
: The busy signal indicates that a trunk or circuit is busy. This is usually signaled by means of a "long" busy tone.

LINEBUSYMODE_UNKNOWN
: The busy signal's specific mode is currently unknown, but may become known later.

LINEBUSYMODE_UNAVAIL
: The busy signal's specific mode is unavailable and will not become known.

LINECALLSTATE_SPECIALINFO
: The call state mode is of type LINESPECIALINFO_. Values are:

LINESPECIALINFO_NOCIRCUIT
: This special information tone preceeds a no circuit or emergency announcement (trunk blockage category).

LINESPECIALINFO_CUSTIRREG
: This special information tone preceeds a vacant number, AIS, Centrex number change and non-working station, access code not dialed or dialed in error, manual intercept operator message (customer irregularity category).

LINESPECIALINFO_REORDER
: This special information tone preceeds a reorder announcement (equipment irregularity category).

LINESPECIALINFO_UNKNOWN
: Specific about the special information tone are currently unknown but may become known later.

LINESPECIALINFO_UNAVAIL
: Specifics about the special information tone are unavailable, and will not become known.

LINECALLSTATE_CONNECTED
: Call state mode is unused.

LINECALLSTATE_PROCEEDING
: Call state mode is unused.

LINECALLSTATE_ONHOLD
: Call state mode is unused.

LINECALLSTATE_CONFERENCED
: Call state mode is unused.

LINECALLSTATE_ONHOLDPENDCONF
: Call state mode is unused.

LINECALLSTATE_ONHOLDPENDTRANSFER
: Call state mode is unused.

LINECALLSTATE_DISCONNECTED
: Call state mode is of type LINEDISCONNECTMODE_. Values are:

LINEDISCONNECTMODE_NORMAL
: This is a "normal" disconnect request by the remote party, the call was terminated normally.

LINEDISCONNECTMODE_UNKNOWN
: The reason for the disconnect request is unknown.

LINEDISCONNECTMODE_REJECT
: The remote user has rejected the call.

LINEDISCONNECTMODE_PICKUP
: The call was picked up from elsewhere.

LINEDISCONNECTMODE_FORWARDED
: The call was forwarded by the switch.

LINEDISCONNECTMODE_BUSY
: The remote user's station is busy.

LINEDISCONNECTMODE_NOANSWER
: The remote user's station does not answer.

LINEDISCONNECTMODE_BADADDRESS
The destination address in invalid.
LINEDISCONNECTMODE_CONGESTION
The network is congested.
LINEDISCONNECTMODE_INCOMPATIBLE
The remote user's station equipment is incompatible for the type of call requested.
LINEDISCONNECTMODE_UNAVAIL
The remote user's station equipment is incompatible for the type of call requested.
LINECALLSTATE_UNKNOWN
Call state mode is unused.

dwCallPrivilege

Specifies the application's privilege for this call, of type LINECALLPRIVILEGE. Values are:

LINECALLPRIVILEGE_MONITOR
The application has monitor privilege.
LINECALLPRIVILEGE_OWNER
The application has owner privilege.

dwCallFeatures

These flags indicate which Telephony API functions can be invoked on the call given the availability of the feature in the device capabilities, the current call state, and call ownership of the invoking application. A zero indicates the corresponding feature cannot be invoked by the application on the call in its current state; a one indicates the feature can be invoked. This field uses the following LINECALLFEATURE_ constants.

dwDevSpecificSize
dwDevSpecificOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device-specific field.

Device-specific extensions should use the DevSpecific (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

The application is sent a **LINE_CALLSTATE** message whenever the call state of a call changes. This message only provides the new call state of the call. Additional status about a call is available through **TSPI_lineGetCallStatus**.

LINEDEVCAPS

[New - Windows 95]

```
typedef struct linedevcaps_tag {
  DWORD  dwTotalSize;
  DWORD  dwNeededSize;
  DWORD  dwUsedSize;

  DWORD  dwProviderInfoSize;
  DWORD  dwProviderInfoOffset;

  DWORD  dwSwitchInfoSize;
  DWORD  dwSwitchInfoOffset;

  DWORD  dwPermanentLineID;
  DWORD  dwLineNameSize;
  DWORD  dwLineNameOffset;
  DWORD  dwStringFormat;
```

```
  DWORD  dwAddressModes;
  DWORD  dwNumAddresses;
  DWORD  dwBearerModes;
  DWORD  dwMaxRate;
  DWORD  dwMediaModes;

  DWORD  dwGenerateToneModes;
  DWORD  dwGenerateToneMaxNumFreq;
  DWORD  dwGenerateDigitModes;
  DWORD  dwMonitorToneMaxNumFreq;
  DWORD  dwMonitorToneMaxNumEntries;
  DWORD  dwMonitorDigitModes;
  DWORD  dwGatherDigitsMinTimeout;
  DWORD  dwGatherDigitsMaxTimeout;

  DWORD  dwMedCtlDigitMaxListSize;
  DWORD  dwMedCtlMediaMaxListSize;
  DWORD  dwMedCtlToneMaxListSize;
  DWORD  dwMedCtlCallStateMaxListSize;

  DWORD  dwDevCapFlags;
  DWORD  dwMaxNumActiveCalls;
  DWORD  dwAnswerMode;
  DWORD  dwRingModes;
  DWORD  dwLineStates;

  DWORD  dwUUIAcceptSize;
  DWORD  dwUUIAnswerSize;
  DWORD  dwUUIMakeCallSize;
  DWORD  dwUUIDropSize;
  DWORD  dwUUISendUserUserInfoSize;
  DWORD  dwUUICallInfoSize;

  LINEDIALPARAMS  MinDialParams;
  LINEDIALPARAMS  MaxDialParams;
  LINEDIALPARAMS  DefaultDialParams;

  DWORD  dwNumTerminals;
  DWORD  dwTerminalCapsSize;
  DWORD  dwTerminalCapsOffset;
  DWORD  dwTerminalTextEntrySize;
  DWORD  dwTerminalTextSize;
  DWORD  dwTerminalTextOffset;

  DWORD  dwDevSpecificSize;
  DWORD  dwDevSpecificOffset;

} LINEDEVCAPS, FAR *LPLINEDEVCAPS;
```

The **LINEDEVCAPS** structure describes the capabilities of a line device.

dwTotalSize
The total size in bytes allocated to this data structure.

dwNeededSize
The size in bytes for this data structure that is needed to hold all the returned information.

dwUsedSize
The size in bytes of the portion of this data structure that contains useful information.

dwProviderInfoSize
dwProviderInfoOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized field containing service provider information.

The **dwProviderInfoSize/Offset** field is intended to provide information about the provider hardware and/or software, such as the vendor name and version numbers of hardware and software. This information can be useful when a user needs to call customer service with problems regarding the provider.

dwSwitchInfoSize
dwSwitchInfoOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device field containing switch information.

The **dwSwitchInfoSize/Offset** field is intended to provide information about the switch to which the line device is connected, such as the switch manaufacturer, the model name, the software version, and so on. This information can be useful when a user needs to call customer service with problems regarding the switch.

dwPermanentLineID

Specifies the permanent DWORD identifier by which the line device is known in the system's configuration. It is a permanent name for the line device. This permanent name (as opposed to *dwDevice ID*) does not change as lines are added and removed from the system. It can therefore be used to link line-specific information in INI files (or other files) in a way that is not affected by adding or removing other lines.

dwLineNameSize
dwLineNameOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device field containing user configurable name for this line device. This name can be configured by the user when configuring the line device's service provider and is provided for the user's convenience.

dwStringFormat

Specifies the string format used with this line device, of type STRINGFORMAT_. Values are:

STRINGFORMAT_ASCII
This is ASCII string format using one byte per character.

STRINGFORMAT_DBCS
This is DBCS string format using two bytes per character.

STRINGFORMAT_UNICODE
This is unicode string format using two bytes per character.

dwAddressModes

Specifies the mode by which the originating address is specified, of type LINEADDRESSMODE_. Values are:

LINEADDRESSMODE_ADDRESSID
The address is specified using a small integer in the range 0 to *dwNumAddresses* minus one, where *dwNumAddresses* is the value in the line's device capabilities.

LINEADDRESSMODE_DIALABLEADDR
The address is specified through its phone number.

dwNumAddresses

Specifies the number of addresses associated with this line device. Individual addresses are referred to by address IDs. Address IDs range from zero to one less than the value indicated by **dwNumAddresses**.

dwBearerModes

This flag array indicates the different bearer modes the address is able to support. This field uses the following LINEBEARERMODE_ constants:

LINEBEARERMODE_VOICE
This is a regular 3.1kHz analog voice grade bearer service. Bit integrity is not assured. Voice can support fax and modem media modes.

LINEBEARERMODE_SPEECH
This corresponds to G.711 speech transmission on the call. The network may use processing techniques such as analog transmission, echo cancellation and compression/decompression. Bit

integrity is not assured. Speech is not intended to support fax and modem media modes.

LINEBEARERMODE_MULTIUSE
The multi-use mode defined by ISDN.

LINEBEARERMODE_DATA
The unrestricted data transfer on the call. The data rate is specified separately.

LINEBEARERMODE_ALTSPEECHDATA
The alternate transfer of speech or unrestricted data on the same call (ISDN).

LINEBEARERMODE_NONCALLSIGNALING
This corresponds to a non-call-associated signaling connection from the application to the service provider or switch (treated as a “media stream” by the Telephony API).

dwMaxRate

This field contains the maximum data rate in bits per second for information exchange over the call.

dwMediaModes

This flag array indicates the different media modes the address is able to support, of type LINEMEDIAMODE_.

LINEMEDIAMODE_UNKNOWN
A media stream exists but its mode is not known. This would correspond to a call with an unclassified media type. In typical analog telephony environments, an inbound call’s media mode may be unknown until after the call has been answered and the media stream filtered to make a determination.

LINEMEDIAMODE_INTERACTIVEVOICE
The presence of voice energy on the call and the call is treated as an interactive call with humans on both ends.

LINEMEDIAMODE_AUTOMATEDVOICE
The presence of voice energy on the call and the voice is locally handled by an automated application.

LINEMEDIAMODE_DATAMODEM
A data modem session on the call.

LINEMEDIAMODE_G3FAX
A group 3 fax is being sent or received over the call.

LINEMEDIAMODE_TDD
A TDD (Telephony Devices for the Deaf) session on the call.

LINEMEDIAMODE_G4FAX
A group 4 fax is being sent or received over the call.

LINEMEDIAMODE_DIGITALDATA
Digital data being transmitted over the call.

LINEMEDIAMODE_TELETEX
A teletex session on the call. Teletex is one of the telematic services.

LINEMEDIAMODE_VIDEOTEX
A videotex session on the call. Videotex is one the telematic services.

LINEMEDIAMODE_TELEX
A telex session on the call. Telex is one the telematic services.

LINEMEDIAMODE_MIXED
A mixed session on the call. Mixed is one the ISDN telematic services.

LINEMEDIAMODE_ADSI
An ADSI (Analog Display Services Interface) session on the call.

dwGenerateToneModes

Specifies the different kinds of tones that can be generated on this line, of type LINETONEMODE_. Values are:

LINETONEMODE_CUSTOM
The tone is a custom tone, defined by the specified frequencies.

LINETONEMODE_RINGBACK
The tone to be generated is ringback tone.

LINETONEMODE_BUSY
The tone is a standard (station) busy tone.

LINETONEMODE_BEEP
The tone is a beep, as used to announce the beginning of a recording.
LINETONEMODE_BILLING
The tone is billing information tone such as a credit card prompt tone.

dwGenerateToneMaxNumFreq
This field contains the maximum number of frequencies that can be specified in describing a general tone using the **LINEGENERATETONE** data structure when generating a tone using **TSPI_lineGenerateTone**. A value of zero indicates that tone generation is not available.

dwGenerateDigitModes
This field specifies the digit modes than can be generated on this line, of type LINEDIGITMODE_. Values are:
LINEDIGITMODE_PULSE
Generate digits as pulse/rotary pulse sequences.
LINEDIGITMODE_DTMF
Generate digits as DTMF tones.

dwMonitorToneMaxNumFreq
This field contains the maximum number of frequencies that can be specified in describing a general tone using the **LINEMONITORTONE** data structure when monitoring a general tone using **TSPI_lineMonitorTones**. A value of zero indicates that tone monitor is not available.

dwMonitorToneMaxNumEntries
This field contains the maximum number of entries that can be specified in a tone list to **TSPI_lineMonitorTones**.

dwMonitorDigitModes
This field specifies the digit modes than can be detected on this line, of type LINEDIGITMODE_. Values are:
LINEDIGITMODE_PULSE
Detect digits as audible clicks that are the result of rotary pulse sequences.
LINEDIGITMODE_DTMF
Detect digits as DTMF tones.
LINEDIGITMODE_DTMFEND
Detect down edges of digits detected as DTMF tones.

dwGatherDigitsMinTimeout
dwGatherDigitsMaxTimeout
These fields contain the minimum and maximum values in milliseconds that can be specified for both the first digit and inter digit timeout values used by **TSPI_lineGatherDigits**. If both these field are zero, timeouts are not supported.

dwMedCtlDigitMaxListSize
dwMedCtlMediaMaxListSize
dwMedCtlToneMaxListSize
dwMedCtlCallStateMaxListSize
These fields contain the maximum number of entries that can be specified in the digit list, the media list, the tone list, and the call state list parameters of **TSPI_lineSetMediaControl** respectively.

dwDevCapFlags
This field specifies various boolean device capabilities, of type LINEDEVCAPFLAGS_. Values are:
LINEDEVCAPFLAGS_CROSSADDRCONF
Specifies whether calls on different addresses on this line can be conferenced.
LINEDEVCAPFLAGS_HIGHLEVCOMP
Specifies whether high-level compatibility information elements are supported on this line.
LINEDEVCAPFLAGS_LOWLEVCOMP
Specifies whether low-level compatibility information elements are supported on this line.
LINEDEVCAPFLAGS_MEDIACONTROL
Specifies whether media control operations are available for calls at this line.
LINEDEVCAPFLAGS_MULTIPLEADDR
Specifies whether **TSPI_lineMakeCall** or **TSPI_lineDial** are able to deal with multiple addresses at once (as for inverse multiplexing).

LINEDEVCAPFLAGS_CLOSEDROP
Specifies what happens when an open line is closed while the application has calls active on the line. If TRUE, a **TSPI_lineClose** will drop (that is, clear) all calls on the line if the application is the sole owner of those calls. Knowing the setting of this flag ahead of time makes it possible for the application to display an OK/Cancel dialog box for the user, warning that the active call will be lost. If CLOSEDROP is FALSE, a **TSPI_lineClose** will not automatically drop any calls still active on the line if the service provider knows that some other device can keep the call alive. For example, if an analog line has the computer and phoneset both connect directly to them (in a party-line configuration), the service provider should set the flag to FALSE, as the offhook phone will automatically keep the call active even after the computer powers down.

LINEDEVCAPFLAGS_DIALBILLING
LINEDEVCAPFLAGS_DIALQUIET
LINEDEVCAPFLAGS_DIALDIALTONE
These flags indicate whether the "$", "@", or "W" dialable string modifier is supported for a given line device. It is TRUE if the modifier is supported; FALSE otherwise. The "?" (prompt user to continue dialing) is never supported by a line device. These flags allow an application to determine "up front" which modifiers would result in the generation of a **LINEERR**. The application has the choice of pre-scanning dialable strings for unsupported characters, or passing the "raw" string from **lineTranslateAddress** directly to the provider as part of **lineMakeCall** (or **lineDial**) and letting the function generate an error to tell it which unsupported modifier occurs first in the string.

dwMaxNumActiveCalls
This field provides the maximum number of (minimum bandwidth) calls that can be active (connected) on the line at any one time. The actual number of active calls may be lower if higher bandwidth calls have been established on the line.

dwAnswerMode
This field specifies the effect on the active call when answering another offering call on a line device, of type LINEANSWERMODE_. Values are:

LINEANSWERMODE_NONE
Answering another call on the same has no effect on the existing active call(s) on the line.

LINEANSWERMODE_DROP
The currently active call will be automatically dropped.

LINEANSWERMODE_HOLD
The currently active call will automatically be placed on hold.

dwRingModes
This field contains the number of different ring modes that can be reported in the **LINE_LINEDEVSTATE** message with the *ringing* indication. Different ring modes range from one to **dwRingModes**. Zero indicates no ring.

dwLineStates
This field specified the different line status components for which the application may be notified in a **LINE_LINEDEVSTATE** message on this line, of type LINEDEVSTATE_. Values are:

LINEDEVSTATE_OTHER
Device-status items other than those listed below have changed.

LINEDEVSTATE_RINGING
The switch tells the line to alert the user.

LINEDEVSTATE_CONNECTED
The line was previously disconnected and is now connected to TAPI.

LINEDEVSTATE_DISCONNECTED
This line was previously connected and is now disconnected from TAPI.

LINEDEVSTATE_MSGWAITON
The "message waiting" indicator is turned on.

LINEDEVSTATE_MSGWAITOFF
The "message waiting" indicator is turned off.

LINEDEVSTATE_INSERVICE
The line is connected to TAPI. This happens when TAPI is first activated, or when the line wire is physically plugged in and in service at the switch while TAPI is active.

LINEDEVSTATE_OUTOFSERVICE
The line is out of service at the switch or physically disconnected. TAPI cannot be used to operate on the line device.

LINEDEVSTATE_MAINTENANCE
Mainatenance is being performed on the line at the switch. TAPI cannot be used to operate on the line device.

LINEDEVSTATE_OPEN
The line has been opened.

LINEDEVSTATE_CLOSE
The line has been closed.

LINEDEVSTATE_NUMCALLS
The number of calls on the line device has changed.

LINEDEVSTATE_NUMCOMPLETIONS
The number of outstanding call completions on the line device has changed.

LINEDEVSTATE_TERMINALS
The terminal settings have changed.

LINEDEVSTATE_ROAMMODE
The roam mode of the line device has changed.

LINEDEVSTATE_BATTERY
The battery level has changed significantly (cellular).

LINEDEVSTATE_SIGNAL
The signal level has changed significantly (cellular).

LINEDEVSTATE_DEVSPECIFIC
The line's device-specific information has changed.

LINEDEVSTATE_REINIT
Items have changed in the configuration of line devices. To become aware of these changes (such as for the appearance of new line devices) the application should reinitialize its use of TAPI. The *hDevice* parameter is left NULL for this state change as it applies to any of the lines in the system.

LINEDEVSTATE_LOCK
The locked status of the line device has changed.

dwUUIAcceptSize
This field specifies the maximum size of user-to-user information that can be sent during a call accept.

dwUUIAnswerSize
This field specifies the maximum size of user-to-user information that can be sent during a call answer.

dwUUIMakeCallSize
This field specifies the maximum size of user-to-user information that can be sent during a make call.

dwUUIDropSize
This field specifies the maximum size of user-to-user information that can be sent during a call drop.

dwUUISendUserUserInfoSize
This field specifies the maximum size of user-to-user information that can be sent separately any time during a call with **TSPI_lineSendUserUserInfo**.

dwUUICallInfoSize
This field specifies the maximum size of user-to-user information that can be received in the **LINECALLINFO** structure.

MinDialParams
MaxDialParams
These fields contain the minimum and maximum values for the dial parameters in milliseconds that can be set for calls on this line. Dialing parameters can be set to values in this range. The granularity of the actual settings are service provider-specific.

DefaultDialParams
This field contains the default dial parameters used for calls on this line. These parameter values

can be overridden on a per-call basis.

dwNumTerminals

Specifies the number of terminals that can be set for this line device, its addresses, or its calls. Individual terminals are referred to by terminal IDs, and range from zero to one less than the value indicated by **dwNumTerminals**.

dwTerminalCapsSize
dwTerminalCapsOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device field containing an array with entries of type **LINETERMCAPS**. This array is indexed by terminal IDs, in the range from zero to **dwNumTerminals** minus one. Each entry in the array specifies the terminal device capabilities of the corresponding terminal.

dwTerminalTextEntrySize

Specifies the size in bytes of each of the terminal text descriptions pointed at by **dwTerminalTextSize/Offset**.

dwTerminalTextSize
dwTerminalTextOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field containing descriptive text about each of the line's available terminals. Each message is **dwTerminalTextEntrySize** bytes long. The string format of these textual descriptions is indicated by **dwStringFormat** in the line's device capabilities.

dwDevSpecificSize
dwDevSpecificOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device-specific field.

Device-specific extensions should use the DevSpecific (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

LINEDEVSTATUS

[New - Windows 95]

```
typedef struct linedevstatus_tag {
  DWORD  dwTotalSize;
  DWORD  dwNeededSize;
  DWORD  dwUsedSize;

  DWORD  dwNumOpens;
  DWORD  dwOpenMediaModes;
  DWORD  dwNumActiveCalls;
  DWORD  dwNumOnHoldCalls;
  DWORD  dwNumOnHoldPendCalls;
  DWORD  dwLineFeatures;
  DWORD  dwNumCallCompletions;
  DWORD  dwRingMode;
  DWORD  dwSignalLevel;
  DWORD  dwBatteryLevel;
  DWORD  dwRoamMode;

  DWORD  dwDevStatusFlags;

  DWORD  dwTerminalModesSize;
  DWORD  dwTerminalModesOffset;

  DWORD  dwDevSpecificSize;
  DWORD  dwDevSpecificOffset;
} LINEDEVSTATUS, FAR *LPLINEDEVSTATUS;
```

The **LINEDEVSTATUS** structure describes the current status of a line device.

dwTotalSize
The total size in bytes allocated to this data structure.

dwNeededSize
The size in bytes for this data structure that is needed to hold all the returned information.

dwUsedSize
The size in bytes of the portion of this data structure that contains useful information.

dwNumOpens
Specifies the number of active opens on the line device.

dwOpenMediaModes
This bit array indicates for which media modes the line device is currently open.

dwNumActiveCalls
The number of calls on the line in call states other than ***idle***, ***onHold***, ***onHoldPendingTransfer***, and *onHoldPendingConference*.

dwNumOnHoldCalls
The number of calls on the line in the *onHold* state.

dwNumOnHoldPendingCalls
The number of calls on the line in the ***onHoldPendingTransfer*** or *onHoldPendingConference* state.

dwLineFeatures
This field specifies the line-related API functions that are currently available on this line, of type LINEFEATURE_. Values are:

LINEFEATURE_DEVSPECIFIC
Device-specific operations can be used on the line.

LINEFEATURE_DEVSPECIFICFEAT
Device-specific features can be used on the line.

LINEFEATURE_FORWARD

Forwarding of all addresses can be used on the line.

LINEFEATURE_MAKECALL
An outbound call can be placed on this line using an unspecified address.

LINEFEATURE_SETMEDIACONTROL
Media control can be set on this line.

LINEFEATURE_SETTERMINAL
Terminal modes for this line can be set.

dwNumCallCompletions
Specifies the number of outstanding call completion requests on the line.

dwRingMode
Specifies the current ring mode on the line device.

dwBatteryLevel
Specifies the current battery level of the line device hardware. This is value in the range 0x00000000 (battery empty) to 0x0000FFFF (battery full).

dwSignalLevel
Specifies the current signal level of the connection on the line. This is value in the range 0x00000000 (weakest signal) to 0x0000FFFF (strongest signal).

dwRoamMode
Specifies the current roam mode of the line device, of type LINEROAMMODE_. Values are:

LINEROAMMODE_UNKNOWN
The roam mode is currently unknown, but may become known later.

LINEROAMMODE_UNAVAIL
The roam mode is unavailable and will not be known.

LINEROAMMODE_HOME
The line is connected to the home network node.

LINEROAMMODE_ROAMA
The line is connected to the Roam-A carrier and calls are charged accordingly.

LINEROAMMODE_ROAMB
The line is connected to the Roam-B carrier and calls are charged accordingly.

dwDevStatusFlags
The size in bytes of this data structure that contains useful information, of type LINEDEVSTATUSFLAGS_. Values are:

LINEDEVSTATUSFLAGS_CONNECTED
Specifies whether the line is connected to TAPI. If TRUE, the line is connected, and API is able to operate on the line device. If FALSE, the line is disconnected, and the application is unable to control the line device through TAPI.

LINEDEVSTATUSFLAGS_MSGWAIT
This field indicates whether the line has a message waiting. If TRUE, a message is waiting; if FALSE, no message is waiting.

LINEDEVSTATUSFLAGS_INSERVICE
This field indicates whether the line is in service. If TRUE, the line is in service; if FALSE, the line is out of service.

LINEDEVSTATUSFLAGS_LOCKED
This field indicates whether the line is locked or unlocked. This bit is most often used with line devices associated with cellular phones. Many cellular phones have a security mechanism which requires the entry of a password to enable the phone to place calls. This bit may be used to indicate to applications that the phone is locked and cannot place calls until the password is entered on the user interface of the phone, so that the application can present an appropriate alert to the user.

dwTerminalModesSize
dwTerminalModesOffset
The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device field containing an array with DWORD-sized entries, of type LINETERMMODE_. This array is indexed by terminal IDs, in the range from zero to **dwNumTerminals** minus one. Each entry in the array specifies the current terminal modes for the corresponding terminal set with the

TSPI_lineSetTerminal operation for this line. Values are:

LINETERMMODE_BUTTONS
: These are button press events sent from the terminal to the line.

LINETERMMODE_LAMPS
: This are lamp events sent from the line to the terminal.

LINETERMMODE_DISPLAY
: This is display information sent from the line to the terminal.

LINETERMMODE_RINGER
: This is ringer control information sent from the switch to the terminal.

LINETERMMODE_HOOKSWITCH
: These are hookswitch event sent between the terminal and the line.

LINETERMMODE_MEDIATOLINE
: This is the unidirectional media stream from the terminal to the line associated with a call on the line. Use this value when routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIAFROMLINE
: This is the unidirectional media stream from the line to the terminal associated with a call on the line. Use this value when routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIABIDIRECT
: This is the bidirectional media stream associated with a call on the line and the terminal. Use this value when routing of both unidirectional channels of a call's media stream cannot be controlled independently.

dwDevSpecificSize
dwDevSpecificOffset
: The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device-specific field.

Device-specific extensions should use the DevSpecific (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

LINEDIALPARAMS

[New - Windows 95]

```
typedef struct linedialparams_tag {
  DWORD  dwDialPause;
  DWORD  dwDialSpeed;
  DWORD  dwDigitDuration;
  DWORD  dwWaitForDialtone;
} LINEDIALPARAMS, FAR *LPLINEDIALPARAMS;
```

The **LINEDIALPARAMS** structure describes a collection of dialing-related parameters.

dwDialPause
Specifies the duration in milliseconds of a comma in the dialable address.

dwDialSpeed
Specifies the inter-digit time period in milliseconds between successive digits.

dwDigitDuration
Specifies the duration in milliseconds of a digit.

dwWaitForDialtone
Specifies the maximum amount of time that should be waited for dialtone when a 'W' is used in the dialable address.

Not extensible.

When a value of zero is specified for a field, the service provider should use the default value for that field. If a non-zero value is specified for a field which is outside the range specified by the corresponding fields in **MinDialParams** and **MaxDialParams** in the **LINEDEVCAPS** structure, the service provider should use the nearest value within the valid range.

The **LINEDIALPARAMS** structure is used to set dialing related parameters on calls. **TSPI_lineMakeCall** allows an application to adjust the dialing parameters to be used for the call. **TSPI_lineSetCallParams** can be used to adjust the dialing parameters of an existing call. The **LINECALLINFO** structure lists the call's current dialing parameters.

LINEEXTENSIONID

[New - Windows 95]

```
typedef struct lineextensionid_tag {
  DWORD  dwExtensionID0;
  DWORD  dwExtensionID1;
  DWORD  dwExtensionID2;
  DWORD  dwExtensionID3;
} LINEEXTENSIONID, FAR *LPLINEEXTENSIONID;
```

The **LINEEXTENSIONID** structure describes an extension ID. Extension IDs are used to identify service provider-specific extensions for line devices.

dwExtensionID0
dwExtensionID1
dwExtensionID2
dwExtensionID3

These four DWORD-sized fields together specify a universally unique extension ID that identifies a line device class extension.

Not extensible.

Extension IDs are generated using an SDK-provided generation utility.

LINEFORWARD

[New - Windows 95]

```
typedef struct lineforward_tag {
  DWORD  dwForwardMode;

  DWORD  dwCallerAddressSize;
  DWORD  dwCallerAddressOffset;

  DWORD  dwDestCountryCode;
  DWORD  dwDestAddressSize;
  DWORD  dwDestAddressOffset;
} LINEFORWARD, FAR *LPLINEFORWARD;
```

The **LINEFORWARD** structure describes an entry of the forwarding instructions.

dwForwardMode

Specifies the types of forwarding, of type LINEFORWARDMODE_. **dwForwardMode** can have only a single bit set. Values are:

LINEFORWARDMODE_UNCOND
: Forward all calls unconditionally irrespective of their origin. Use this value when unconditional forwarding for internal and external calls cannot be controlled separately. Unconditional forwarding overrides forwarding on busy and/or no answer conditions.

LINEFORWARDMODE_UNCONDINTERNAL
: Forward all internal calls unconditionally. Use this value when unconditional forwarding for internal and external calls can be controlled separately.

LINEFORWARDMODE_UNCONDEXTERNAL
: Forward all external calls unconditionally. Use this value when unconditional forwarding for internal and external calls can be controlled separately.

LINEFORWARDMODE_UNCONDSPECIFIC
: Forward all calls that originated at a specified address unconditionally (selective call forwarding).

LINEFORWARDMODE_BUSY
: Forward all calls on busy irrespective of their origin. Use this value when forwarding for internal and external calls on busy and no answer cannot be controlled separately.

LINEFORWARDMODE_BUSYINTERNAL
: Forward all internal calls on busy. Use this value when forwarding for internal and external calls on busy and no answer can be controlled separately.

LINEFORWARDMODE_BUSYEXTERNAL
: Forward all external calls on busy. Use this value when forwarding for internal and external calls on busy and no answer can be controlled separately.

LINEFORWARDMODE_BUSYSPECIFIC
: Forward all calls that originated at a specified address on busy (selective call forwarding).

LINEFORWARDMODE_NOANSW
: Forward all calls on no answer irrespective of their origin. Use this value when call forwarding for internal and external calls on no answer cannot be controlled separately.

LINEFORWARDMODE_NOANSWINTERNAL
: Forward all internal calls on no answer. Use this value when forwarding for internal and external calls on no answer can be controlled separately.

LINEFORWARDMODE_NOANSWEXTERNAL
: Forward all external calls on no answer. Use this value when forwarding for internal and external calls on no answer can be controlled separately.

LINEFORWARDMODE_NOANSWSPECIFIC
: Forward all calls that originated at a specified address on no answer (selective call forwarding).

LINEFORWARDMODE_BUSYNA
: Forward all calls on busy/no answer irrespective of their origin. Use this value when forwarding for internal and external calls on busy and no answer cannot be controlled separately.

LINEFORWARDMODE_BUSYNAINTERNAL
: Forward all internal calls on busy/no answer. Use this value when call forwarding on busy and no

answer cannot be controlled separately for internal calls.

LINEFORWARDMODE_BUSYNAEXTERNAL
Forward all external calls on busy/no answer. Use this value when call forwarding on busy and no answer cannot be controlled separately for internal calls.

LINEFORWARDMODE_BUSYNASPECIFIC
Forward all calls that originated at a specified address on busy/no answer (selective call forwarding).

dwCallerAddressSize
dwCallerAddressOffset

The size in bytes and the offset from the beginning of the containing data structure in bytes of the variably sized address field containing the address of a caller to be forwarded.
dwCallerAddressSize/Offset is set to zero if **dwForwardMode** is not one of the following: LINEFORWARDMODE_BUSYNASPECIFIC, LINEFORWARDMODE_NOANSWSPECIFIC, LINEFORWARDMODE_UNCONDSPECIFIC, or LINEFORWARDMODE_BUSYSPECIFIC.

The offset **dwCallerAddressOffset** in the variably sized field of type **LINEFORWARD** pointed at by **dwForwardSize/Offset** is relative to the beginning of the **LINEADDRESSSTATUS** data structure (the “root” container).

dwDestCountryCode

Specifies the country code of the destination address where the call is to be forwarded to.

dwDestAddressSize
dwDestAddressOffset

The size in bytes and the offset from the beginning of the containing data structure in bytes of the variably sized address field containing the address of the address where calls are to be forwarded to.

The offset **dwDestAddressOffset** in the variably sized field of type **LINEFORWARD** pointed at by **dwForwardSize/Offset** is relative to the beginning of the **LINEADDRESSSTATUS** data structure (the “root” container).

Not extensible.

Each entry in the **LINEFORWARD** structure specifies a forwarding request.

LINEFORWARDLIST

[New - Windows 95]

```
typedef struct lineforwardlist_tag {
  DWORD  dwTotalSize;

  DWORD  dwNumEntries;
  LINEFORWARD  ForwardList[1];
} LINEFORWARDLIST, FAR *LPLINEFORWARDLIST;
```

The **LINEFORWARDLIST** structure describes a list of forwarding instructions.

dwTotalSize
Specifies the total size in bytes of the data structure.

dwNumEntries
Specifies number of entries in the array specified as **ForwardList[ ]**.

ForwardList[ ]
Specifies an array of forwarding instruction. The array's entries are of type **LINEFORWARD**.

Not extensible.

The **LINEFORWARDLIST** structure defines the forwarding parameters requested for forwarding calls on an address or all addresses on a line.

LINEGENERATETONE

[New - Windows 95]

```
typedef struct linegeneratetone_tag {
  DWORD  dwFrequency;
  DWORD  dwCadenceOn;
  DWORD  dwCadenceOff;
  DWORD  dwVolume;
} LINEGENERATETONE, FAR *LPLINEGENERATETONE;
```

The **LINEGENERATETONE** structure contains information about a tone to be generated.

dwFrequency
Specifies the frequency in Hertz of this tone component. A service provider may adjust (round up or down) the frequency specified by the application to fit its resolution.

dwCadenceOn
Determines the “on” duration in milliseconds of the cadence of the custom tone to be generated. Zero means no tone is generated.

dwCadenceOff
Determines the “off” duration in milliseconds of the cadence of the custom tone to be generated. Zero means no off time - a constant tone.

dwVolume
Determines the volume level at which the tone is to be generated. A value of 0x0000FFFF represents full volume, and a value of 0x00000000 is silence.

Not extensible.

This structure is only used for the generation of tones, it is not used for tone monitoring.

LINEMEDIACONTROLCALLSTATE

[New - Windows 95]

```
typedef struct linemediacontrolcallstate_tag {
  DWORD  dwCallStates;
  DWORD  dwMediaControl;
} LINEMEDIACONTROLCALLSTATE,
FAR *LPLINEMEDIACONTROLCALLSTATE;
```

The **LINEMEDIACONTROLCALLSTATE** structure describes a media action to be executed when detecting transitions into one or more call states.

dwCallStates

Specifies one or more call states, of type LINECALLSTATE_. Values are:

LINECALLSTATE_IDLE
The call is idle—no call actually exists.

LINECALLSTATE_OFFERING
The call is being offered to the station, signaling the arrival of a new call. In some environments, a call in the *offering* state does not automatically alert the user; alerting is done by the switch instructing the line to ring, it does not affect any call states.

LINECALLSTATE_ACCEPTED
The call was offering and has been accepted. This indicates to other (monitoring) applications that the current owner application has claimed responsibility for answering the call. In ISDN, this also initiates alerting to both parties.

LINECALLSTATE_DIALTONE
The call is receiving a dial tone from the switch, which means that the switch is ready to receive a dialed number.

LINECALLSTATE_DIALING
Destination address information (a phone number) is being sent to the switch over the call. Note that the operation **TSPI_lineGenerateDigits** does not place the line into the *dialing* state.

LINECALLSTATE_RINGBACK
The call is receiving ringback from the called address. Ringback indicates that the other station has been reached and is being alerted.

LINECALLSTATE_BUSY
The call is receiving a busy tone. Busy tone indicates that the call cannot be completed because either a circuit (trunk) or the remote party's station are in use.

LINECALLSTATE_SPECIALINFO
Special information is sent by the network. Special information is typically sent when the destination cannot be reached.

LINECALLSTATE_CONNECTED
The call has been established, the connection is made. Information is able to flow over the call between the originating address and the destination address.

LINECALLSTATE_PROCEEDING
Dialing has completed and the call is proceeding through the switch or telephone network.

LINECALLSTATE_ONHOLD
The call is on hold by the switch.

LINECALLSTATE_CONFERENCED
The call is currently a member of a multi-party conference call.

LINECALLSTATE_ONHOLDPENDCONF
The call is currently on hold while it is being added to a conference.

LINECALLSTATE_ONHOLDPENDTRANSFER
The call is currently on hold awaiting transfer to another number.

LINECALLSTATE_DISCONNECTED
The remote party has disconnected from the call.

LINECALLSTATE_UNKNOWN
The state of the call is not known. This may be due to limitations of the call progress detection implementation.

dwMediaControl

Specifies the media control action, of type LINEMEDIACONTRO**L**. Values are:

LINEMEDIACONTROL_NONE
No change is to be made to the media stream.

LINEMEDIACONTROL_START
Start the media stream.

LINEMEDIACONTROL_RESET
Reset the media stream. Provide the effect of an end-of-input. All buffers are released.

LINEMEDIACONTROL_PAUSE
Temporarily pause the media stream.

LINEMEDIACONTROL_RESUME
Start or resume a paused media stream.

LINEMEDIACONTROL_RATEUP
The speed of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_RATEDOWN
The speed of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_RATENORMAL
The speed of the media stream is returned to normal.

LINEMEDIACONTROL_VOLUMEUP
The amplitude of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMEDOWN
The amplitude of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMENORMAL
The amplitude of the media stream is returned to normal.

Not extensible.

The **LINEMEDIACONTROLCALLSTATE** structure defines a triple <call state(s), media control action>. An array of these triples is passed to the **TSPI_lineSetMediaControl** function to set media control actions triggered by the transition to the call state of the given call. When a transition to a listed call state is detected, the corresponding action on the media stream is invoked.

LINEMEDIACONTROLDIGIT

[New - Windows 95]

```
typedef struct linemediacontroldigit_tag {
  DWORD      dwDigit;
  DWORD      dwDigitModes;
  DWORD      dwMediaControl;
} LINEMEDIACONTROLDIGIT, FAR *LPLINEMEDIACONTROLDIGIT;
```

The **LINEMEDIACONTROLDIGIT** structure describes a media action to be executed when detecting a digit. It is used as as entry in an array.

dwDigit

The low-order byte of this DWORD specifies the digit in ASCII whose detection is to trigger a media action. Valid digits depend on the media mode.

dwDigitModes

Specifies the digit mode(s) that are to be monitored, of type LINEDIGITMODE_. Values are:

LINEDIGITMODE_PULSE
: Detect digits as audible clicks that are the result of rotary pulse sequences. Valid digits for pulse are ‘0’ through ‘9’.

LINEDIGITMODE_DTMF
: Detect digits as DTMF tones. Valid digits for DTMF are ‘0’ through ‘9’, ‘A’, ‘B’, ‘C’, ‘D’, ‘*’, and ‘#’.

LINEDIGITMODE_DTMFEND
: Detect and provide application notification of DTMF down edges. Valid digits for DTMF are ‘0’ through ‘9’, ‘A’, ‘B’, ‘C’, ‘D’, ‘*’, and ‘#’.

dwMediaControl

Specifies the media control action, of type LINEMEDIACONTROL_. Values are:

LINEMEDIACONTROL_NONE
: No change is to be made to the media stream.

LINEMEDIACONTROL_START
: Start the media stream.

LINEMEDIACONTROL_RESET
: Reset the media stream. Provide the effect of an end-of-input. All buffers are released.

LINEMEDIACONTROL_PAUSE
: Temporarily pause the media stream.

LINEMEDIACONTROL_RESUME
: Start or resume a paused media stream.

LINEMEDIACONTROL_RATEUP
: The speed of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_RATEDOWN
: The speed of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_RATENORMAL
: The speed of the media stream is returned to normal.

LINEMEDIACONTROL_VOLUMEUP
: The amplitude of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMEDOWN
: The amplitude of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMENORMAL
: The amplitude of the media stream is returned to normal.

Not extensible.

The **LINEM EDIACONTROLMEDIA** structure defines a triple <digit, digit mode(s), media control action>. An array of these triples is passed to the **TSPI_lineSetMediaControl** function to set media control actions triggered by digits detected on a given call. When a listed digit is detected, the corresponding action on the media stream is invoked.

LINEMEDIACONTROLMEDIA

[New - Windows 95]

```
typedef struct linemediacontrolmedia_tag {
  DWORD      dwMediaModes;
  DWORD      dwDuration;
  DWORD      dwMediaControl;
} LINEMEDIACONTROLMEDIA, FAR *LPLINEMEDIACONTROLMEDIA;
```

The **LINEMEDIACONTROLMEDIA** structure describes a media action to be executed when detecting a media mode change. It is used as as entry in an array.

dwMediaModes

Specifies one or more media modes, of type LINEMEDIAMODE_. Values are:

LINEMEDIAMODE_INTERACTIVEVOICE
The presence of voice energy on the call and the call is treated as an interactive call with humans on both ends.

LINEMEDIAMODE_AUTOMATEDVOICE
The presence of voice energy on the call and the voice is locally handled by an automated application.

LINEMEDIAMODE_DATAMODEM
A data modem session on the call.

LINEMEDIAMODE_G3FAX
A group 3 fax is being sent or received over the call.

LINEMEDIAMODE_TDD
A TDD (Telephony Devices for the Deaf) session on the call.

LINEMEDIAMODE_G4FAX
A group 4 fax is being sent or received over the call.

LINEMEDIAMODE_DIGITALDATA
Digital data being sent or received over the call.

LINEMEDIAMODE_TELETEX
A teletex session on the call. Teletex is one of the telematic services.

LINEMEDIAMODE_VIDEOTEX
A videotex session on the call. Videotex is one the telematic services.

LINEMEDIAMODE_TELEX
A telex session on the call. Telex is one the telematic services.

LINEMEDIAMODE_MIXED
A mixed session on the call. Mixed is one the ISDN telematic services.

LINEMEDIAMODE_ADSI
An ADSI (Analog Display Services Interface) session on the call.

dwDuration

Specifies the duration in milliseconds during which the media mode should be present before the application should be notified or media control action should be taken.

dwMediaControl

Specifies the media control action, of type LINEMEDIACONTROL_. Values are:

LINEMEDIACONTROL_NONE
No change is to be made to the media stream.

LINEMEDIACONTROL_START
Start the media stream.

LINEMEDIACONTROL_RESET
Reset the media stream. Provide the effect of an end-of-input. All buffers are released.

LINEMEDIACONTROL_PAUSE
Temporarily pause the media stream.

LINEMEDIACONTROL_RESUME
Start or resume a paused media stream.

LINEMEDIACONTROL_RATEUP

The speed of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_RATEDOWN

The speed of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_RATENORMAL

The speed of the media stream is returned to normal.

LINEMEDIACONTROL_VOLUMEUP

The amplitude of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMEDOWN

The amplitude of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMENORMAL

The amplitude of the media stream is returned to normal.

Not extensible.

The **LINEMEDIACONTROLMEDIA** structure defines a triple <media mode(s), duration, media control action>. An array of these triples is passed to the **TSPI_lineSetMediaControl** function to set media control actions triggered by media mode changes for a given call. When a change to a listed media mode is detected, the corresponding action on the media stream is invoked.

LINEMEDIACONTROLTONE

[New - Windows 95]

```
typedef struct linemediacontroltone_tag {
  DWORD  dwAppSpecific;
  DWORD  dwDuration;
  DWORD  dwFrequency1;
  DWORD  dwFrequency2;
  DWORD  dwFrequency3;
  DWORD  dwMediaControl;
} LINEMEDIACONTROLTONE, FAR *LPLINEMEDIACONTROLTONE;
```

The **LINEMEDIACONTROLTONE** structure describes a media action to be executed when detecting a tone has been detected. It is used as as entry in an array.

dwAppSpecific
This field is used by the application for tagging the tone. When this tone is detected, the value of the **dwAppSpecific** field is passed back to the application.

dwDuration
Specifies the duration in milliseconds during which the tone should be present before a detection is made.

dwFrequency1
dwFrequency2
dwFrequency3
Specifies the frequency in Hertz of a component of the tone. If fewer than three frequencies are needed in the tone, a value of zero should be used for the unused frequencies. A tone with all three frequencies set to zero is interpreted as silence, and can be use for silence detection.

dwMediaControl
Specifies the media control action, of type LINEMEDIACONTROL_. Values are:

LINEMEDIACONTROL_NONE
No change is to be made to the media stream.

LINEMEDIACONTROL_START
Start the media stream.

LINEMEDIACONTROL_RESET
Reset the media stream. Provide the effect of an end-of-input. All buffers are released.

LINEMEDIACONTROL_PAUSE
Temporarily pause the media stream.

LINEMEDIACONTROL_RESUME
Start or resume a paused media stream.

LINEMEDIACONTROL_RATEUP
The speed of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_RATEDOWN
The speed of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_RATENORMAL
The speed of the media stream is returned to normal.

LINEMEDIACONTROL_VOLUMEUP
The amplitude of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMEDOWN
The amplitude of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMENORMAL
The amplitude of the media stream is returned to normal.

Not extensible.

The **LINEMEDIACONTROLTONE** structure defines a tuple <tone, media control action>. An array of these tuples is passed to the **TSPI_lineSetMediaControl** function to set media control actions triggered by media mode changes for a given call. When a change to a listed media mode is detected, the corresponding action on the media stream is invoked.

A tone with all frequencies set to zero corresponds to silence. An application can thus monitor the call's information stream for silence.

LINEMONITORTONE

[New - Windows 95]

```
typedef struct linemonitortone_tag {
  DWORD  dwAppSpecific;
  DWORD  dwDuration;
  DWORD  dwFrequency1;
  DWORD  dwFrequency2;
  DWORD  dwFrequency3;
} LINEMONITORTONE, FAR *LPLINEMONITORTONE;
```

The **LINEMONITORTONE** structure describes a tone to be monitored. This is used as an entry in an array.

dwAppSpecific

This field is used by the application for tagging the tone. When this tone is detected, the value of the **dwAppSpecific** field is passed back to the application.

dwDuration

Specifies the duration in milliseconds during which the tone should be present before a detection is made.

dwFrequency1
dwFrequency2
dwFrequency3

Specifies the frequency in Hertz of a component of the tone. If fewer than three frequencies are needed in the tone, a value of zero should be used for the unused frequencies. A tone with all three frequencies set to zero is interpreted as silence, and can be use for silence detection.

Not extensible.

The **LINEMONITORTONE** structure defines a tone for the purpose of detection. An array of tones is passed to the **TSPI_lineMonitorTones** function which monitors these tones and send a **LINE_MONITORTONE** message to the application when a detection is made.

A tone with all frequencies set to zero corresponds to silence. An application can thus monitor the call's information stream for silence.

LINETERMCAPS

[New - Windows 95]

```
typedef struct linetermcaps_tag {
  DWORD  dwTermDev;
  DWORD  dwTermModes;
  DWORD  dwTermSharing;
} LINETERMCAPS, FAR *LPLINETERMCAPS;
```

The **LINETERMCAPS** structure describes the capabilities of a line's terminal device.

dwTermDev

Specifies the device type of the terminal, of type LINETERMDEV_. Values are:

LINETERMDEV_PHONE
The terminal is a phone set.

LINETERMDEV_HEADSET
The terminal is a headset

LINETERMDEV_SPEAKER
The terminal is an external speaker and microphone.

dwTermModes

Specifies the terminal mode(s) the terminal device is able to deal with, of type LINETERMMODE_. Values are:

LINETERMMODE_BUTTONS
These are button press events sent from the terminal to the line.

LINETERMMODE_LAMPS
This are lamp events sent from the line to the terminal.

LINETERMMODE_DISPLAY
This is display information sent from the line to the terminal.

LINETERMMODE_RINGER
This is ringer control information sent from the switch to the terminal.

LINETERMMODE_HOOKSWITCH
These are hookswitch event sent from the terminal to the line.

LINETERMMODE_MEDIATOLINE
This is the unidirectional media stream from the terminal to the line associated with a call on the line. Use this value when routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIAFROMLINE
This is the unidirectional media stream from the line to the terminal associated with a call on the line. Use this value when routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIABIDIRECT
This is the bidirectional media stream associated with a call on the line and the terminal. Use this value when routing of both unidirectional channels of a call's media stream cannot be controlled independently.

dwTermSharing

Specifies how the terminal device is shared between line devices, of type LINETERMSHARING_. Values are:

LINETERMSHARING_PRIVATE
The terminal device is private to a single line device.

LINETERMSHARING_SHAREDEXCL
The terminal device can be used by multiple lines. The last line device to do a **TSPI_lineSetTerminal** to the terminal for a given terminal mode will have exclusive connection to the terminal for that mode.

LINETERMSHARING_SHAREDCONF
The terminal device can be used by multiple lines. The **TSPI_lineSetTerminal** requests of the various terminals end up being "merged" at the terminal.

Not extensible.

Line Device Constants

Line Device Constants

Line Device Constants

About Line Device Constants

Introduction

This chapter describes the line-device constants defined by the Telephony SPI.

Extensibility

Provisions are made for extending constants and structures both in a device independent way and in a device-specific (vendor-specific) way.

In constants that are scalar enumerations, a range of values is reserved for future common extensions. The remainder of values is identified as device specific. A vendor can define meanings for these values in any way desired. The interpretation of these values is keyed to the *extension ID* provided via the **LINEDEVCAPS** data structure. For constants that are defined as bit flags, a range of low-order bits are reserved, where the high-order bits can be extension specific. It is recommended that extended values and bit arrays use bits from the highest value or high-order bit down. This leaves the option to move the border between the common portion and extension portion if there is need to do so in the future. Extensions to data structures are assigned a variably sized field with size/offset being part of the fixed part. The TSPI describes for each data structures what device-specific extensions are allowed.

In addition to recognizing a specific extension ID, TAPI.DLL (operating on behalf of an application) must negotiate the extension version number that the application and the service provider will operate under. This is done using the **TSPI_lineNegotiateExtVersion** and **TSPI_phoneNegotiateExtVersion** functions.

An extension ID is a globally unique identifier. There is no central registry for extension IDs. Instead, they are generated locally by the manufacturer by a utility that is available with the toolkit. To guarantee global uniqueness, the number is made up parts such as a (unique) LAN address, time of day, and a random number. Globally Unique Identifiers are designed to be distinguishable from HP/DEC universally unique identifiers and are thus fully compatible with them.

Reference

LINEADDRCAPFLAGS_ Constants

[New - Windows 95]

The LINEADDRCAPFLAGS__bit-flag constants are used in the **dwAddrCapFlags** field of the **LINEADDRESSCAPS** data structure to describe various Boolean address capabilities.

LINEADDRCAPFLAGS_FWDNUMRINGS
Specifies whether the number of rings for a no answer can be specified when forwarding calls on no answer. If TRUE, the valid range is provided in **LINEADDRESSCAPS**.

LINEADDRCAPFLAGS_PICKUPGROUPID
Specifies whether a group ID is required for call pickup.

LINEADDRCAPFLAGS_SECURE
Specifies whether calls on this address can be made secure at call setup time.

LINEADDRCAPFLAGS_BLOCKIDDEFAULT
Specifies whether the network by default sends or blocks caller ID information when making a call on this address. If TRUE, ID information is blocked by default; if FALSE, ID information is transmitted by default.

LINEADDRCAPFLAGS_BLOCKIDOVERRIDE
Specifies whether the default setting for sending or blocking of caller ID information can be overridden per call. If TRUE, override is possible; if FALSE, override is not possible.

LINEADDRCAPFLAGS_DIALED
Specifies whether a destination address can be dialed on this address for making a call. TRUE if a destination address must be dialed; FALSE if the destination address is fixed (such as with a “hot phone”).

LINEADDRCAPFLAGS_ORIGOFFHOOK
Specifies whether the originating party’s phone can automatically be taken offhook when making calls.

LINEADDRCAPFLAGS_DESTOFFHOOK
Specifies whether the called party’s phone can automatically be forced offhook when making calls.

LINEADDRCAPFLAGS_FWDCONSULT
Specifies whether call forwarding involves the establishment of a consultation call.

LINEADDRCAPFLAGS_SETUPCONFNULL
Specifies whether setting up a conference call starts out with an initial call (FALSE) or with no initial call (TRUE).

LINEADDRCAPFLAGS_AUTORECONNECT
Specifies whether dropping a consultation call automatically reconnects to the call on consultation hold. TRUE if reconnect happens automatically, FALSE otherwise.

LINEADDRCAPFLAGS_COMPLETIONID
Specifies whether the completion IDs returned by **TSPI_lineCompleteCall** are useful and unique. TRUE is useful; FALSE otherwise.

LINEADDRCAPFLAGS_TRANSFERHELD
Specifies whether a (hard) held call can be transferred. Often, only calls on consultation hold may transferred.

LINEADDRCAPFLAGS_TRANSFERMAKE
Specifies whether an entirely new call can be established for use as a consultation call on transfer.

LINEADDRCAPFLAGS_CONFERENCEHELD
Specifies whether a (hard) held call can be conferenced to. Often, only calls on consultation hold may be able to be added to as a conference call.

LINEADDRCAPFLAGS_CONFERENCEMAKE
Specifies whether an entirely new call can be established for use as a consultation call (to add) on conference.

LINEADDRCAPFLAGS_PARTIALDIAL
Specifies whether partial dialing is available.

LINEADDRCAPFLAGS_FWDSTATUSVALID

Specifies whether the forwarding status in the **LINEADDRESSSTATUS** structure for this address is valid or is at most a “best estimate” given absence of accurate confirmation by the switch or network.

LINEADDRCAPFLAGS_FWDINTEXTADDR

Specifies whether internal and external calls can be forwarded to different forwarding addresses. This flag is meaningful only if forwarding of internal and external calls can be controlled separately. This flag is TRUE if internal and external calls can be forwarded to different destination addresses; FALSE otherwise.

LINEADDRCAPFLAGS_FWDBUSYNAADDR

Specifies whether call forwarding for busy and no answer can use different forwarding addresses. This flag is meaningful only if forwarding for busy and no answer can be controlled separately. This flag is TRUE if forwarding for busy and no answer can use different destination addresses; FALSE otherwise.

LINEADDRCAPFLAGS_ACCEPTTOALERT

This flag has the value of TRUE if an offering call must be accepted using **lineAccept** to start alerting the users at both ends of the call; Otherwise, it is FALSE. This flag is typically only used with ISDN.

LINEADDRCAPFLAGS_CONFDROP

This flag has the value of TRUE if **lineDrop** of a conference call parent also has the side effect of dropping (disconnecting) the other parties involved in the conference call. It is FALSE if dropping a conference call still allows the other parties to talk among themselves.

LINEADDRCAPFLAGS_PICKUPCALLWAIT

This flag has the value of TRUE if **linePickup** can be used to pick up a call detected by the user as a call waiting call. Otherwise, it is FALSE.

Not extensible. All 32 bits are reserved.

LINEADDRESSMODE_ Constants

[New - Windows 95]

The LINEADDRESSMODE_ bit-flag constants describe various ways of identifying an address on a line device.

LINEADDRESSMODE_ADDRESSID

The address is specified using a small integer in the range 0 to *dwNumAddresses* minus one, where *dwNumAddresses* is the value in the line's device capabilities.

LINEADDRESSMODE_DIALABLEADDR

The address is specified through its phone number.

The high-order 16 bits can be assigned for device-specific extensions. The low-order 16 bits are reserved.

This data type is used to select an address on a line on which to originate a call. The usual model would select the address by means of its address ID. Address IDs are the mechanism used to identify addresses throughout TAPI. However, in some environments, when making a call, it is often more practical to identify a call's originating address by phone number rather than by address ID. One example is in the possible modeling of large numbers of stations (third party) on the switch by means of one line device with lots of addresses. The line represents the set of all stations, and each station is mapped to an address with its own primary phone number and address ID.

LINEADDRESSSHARING_ Constants

[New - Windows 95]

The LINEADDRESSSHARING_ bit-flag constants describe various ways an address can be shared between lines.

LINEADDRESSSHARING_PRIVATE
: The address is private to the user's line, it is not assigned to any other station.

LINEADDRESSSHARING_BRIDGEDEXCL
: The address is bridged to one or more other stations. The first line to activate a call on the line will have exclusive access to the address and calls that may exist on it. Other lines will not be able to use the bridged address while it is in use.

LINEADDRESSSHARING_BRIDGEDNEW
: The address is bridged with one or more other stations. The first line to activate a call on the line will have exclusive access to only the corresponding call. Other applications that use the address will result in new and separate call appearances.

LINEADDRESSSHARING_BRIDGEDSHARED
: The address is bridged with one or more other lines. All bridged parties can share in calls on the address, which then functions as a conference.

LINEADDRESSSHARING_MONITORED
: This is an address whose idle/busy status is made available to this line.

Not extensible. All 32 bits are reserved.

The way in which an address is shared across lines can affect the behavior of that address. **LINE_CALLSTATE** and **LINE_ADDRESSSTATE** messages are sent to the application in response to activities by the bridging stations. It is in these messages that an application can track the status of the address.

LINEADDRESSSTATE_ Constants

[New - Windows 95]

The LINEADDRESSSTATE_ bit-flag constants describe various address status items.

LINEADDRESSSTATE_OTHER
: Address-status items other than those listed below have changed.

LINEADDRESSSTATE_DEVSPECIFIC
: The device-specific item of the address status has changed.

LINEADDRESSSTATE_INUSEZERO
: The address has changed to idle (it is not in use by any stations).

LINEADDRESSSTATE_INUSEONE
: The address has changed from being idle or from being in use by many bridged stations to being in use by just one station.

LINEADDRESSSTATE_INUSEMANY
: The monitored or bridged address has changed to being in use by one station to being used by more than one station.

LINEADDRESSSTATE_NUMCALLS
: The number of calls on the address has changed. This is the result of such things as a new inbound call, an outbound call on the address, or a call changing its hold status. If this bit is set, one or more of the following fields in the **LINEADDRESSSTATUS** structure has changed: **dwNumActiveCalls**, **dwNumOnHoldCalls** and **dwNumOnHoldPendingCalls**. The application should check all three fields when it receives a **LINE_ADDRESSSTATE** (*numCalls*) message.

LINEADDRESSSTATE_FORWARD
: The forwarding status of the address has changed, including possibly the number of rings for determining a no answer condition. The application should check the address status to determine details about the address's current forwarding status.

LINEADDRESSSTATE_TERMINALS
: The terminal settings for the address have changed.

Not extensible. All 32 bits are reserved.

An application is notified about changes to these status items in the **LINE_ADDRESSSTATE** message. The address's device capabilities indicate which address state changes can possibly be reported for this address.

LINEADDRFEATURE_ Constants

[New - Windows 95]

The LINEADDRFEATURE_ constants list the operations that can be invoked on an address using this API.

LINEADDRFEATURE_FORWARD
The address can be forwarded.

LINEADDRFEATURE_MAKECALL
An outbound call can placed on the address.

LINEADDRFEATURE_PICKUP
A call can be picked up at the address.

LINEADDRFEATURE_SETMEDIACONTROL
Media control can be set on this address.

LINEADDRFEATURE_SETTERMINAL
The terminal modes for this address can be set.

LINEADDRFEATURE_SETUPCONF
A conference call with a NULL initial call can be set up at this address.

LINEADDRFEATURE_UNCOMPLETECALL
Call completion requests can be canceled at this address.

LINEADDRFEATURE_UNPARK
Calls can be unparked using this address.

Not extensible. All 32 bits are reserved.

This data type is used both in **LINEADDRESSCAPS** (returned by **TSPI_lineGetAddressCaps**) and in **LINEADDRESSSTATUS** (returned by **TSPI_lineGetAddressStatus**). **LINEADDRESSCAPS** reports the availability of the address features by the service provider (mainly the switch) for a given address. An application would make this determination when it initializes. **LINEADDRESSSTATUS** reports, for a given address, which address features can actually be invoked while the address is in the current state. An application would make this determination dynamically, after address state changes that are typically caused by call-related activities on the address.

LINEANSWERMODE_ Constants

[New - Windows 95]

The LINEANSWERMODE_ bit-flag constants describe how an existing active call on a line device is affected by answering another offering call on the same line.

LINEANSWERMODE_NONE

Answering another call on the same line has no effect on the existing active call(s) on the line.

LINEANSWERMODE_DROP

The currently active call will automatically be dropped.

LINEANSWERMODE_HOLD

The currently active call will automatically be placed on hold.

Not extensible. All 32 bits are reserved.

If a call comes in (is offered) at the time another call is already active, the new call is connected to by invoking **TSPI_lineAnswer**. The effect this has on the existing active call depends on the line's device capabilities. The first call may be unaffected, it may automatically be dropped, or it may automatically be placed on hold.

LINEBEARERMODE_ Constants

[New - Windows 95]

The LINEBEARERMODE_ bit-flag constants describe different bearer modes of a call. When an application makes a call, it can request a specific bearer mode. These modes are used to select a certain quality of service for the requested connection from the underlying telephone network. Bearer modes available on a given line are a device capability of the line.

LINEBEARERMODE_VOICE
: This is a regular 3.1kHz analog voice grade bearer service. Bit integrity is not assured. Voice can support fax and modem media modes.

LINEBEARERMODE_SPEECH
: This corresponds to G.711 speech transmission on the call. The network may use processing techniques such as analog transmission, echo cancellation and compression/decompression. Bit integrity is not assured. Speech is not intended to support fax and modem media modes.

LINEBEARERMODE_MULTIUSE
: The multi-use mode defined by ISDN.

LINEBEARERMODE_DATA
: The unrestricted data transfer on the call. The data rate is specified separately.

LINEBEARERMODE_ALTSPEECHDATA
: The alternate transfer of speech or unrestricted data on the same call (ISDN).

LINEBEARERMODE_NONCALLSIGNALING
: This corresponds to a non-call-associated signaling connection from the application to the service provider or switch (treated as a “media stream” by the Telephony API).

The high-order 16 bits can be assigned for device-specific extensions. The low-order 16 bits are reserved.

Note that bearer mode and media mode are different notions. The bearer mode of a call is an indication of the quality of the telephone connection as provided primarily by the network. The media mode of a call is an indication of the type of information stream that is exchanged over that call. Group 3 fax or data modem are media modes that use a call with a 3.1kHz voice bearer mode.

LINEBUSYMODE_ Constants

[New - Windows 95]

The LINEBUSYMODE_ bit-flag constants describe different busy signals that the switch or network may generate. The different busy signals typically indicate that a different resource required to make a call is currently in use.

LINEBUSYMODE_STATION

The busy signal indicates that the called party's station is busy. This is usually signaled with a "normal" busy tone.

LINEBUSYMODE_TRUNK

The busy signal indicates that a trunk or circuit is busy. This is usually signaled with a "fast" busy tone.

LINEBUSYMODE_UNKNOWN

The busy signal's specific mode is currently unknown, but may become known later.

LINEBUSYMODE_UNAVAIL

The busy signal's specific mode is unavailable and will not become known.

The high-order 16 bits can be assigned for device-specific extensions. The low-order 16 bits are reserved.

Note that busy signals may be sent as inband tones or out-of-band messages. TAPI makes no assumption about the specific signaling mechanism.

LINECALLCOMPLCOND_ Constants

[New - Windows 95]

The LINECALLCOMPLCOND_ bit-flag constants describe different ways in which a call can be completed.

LINECALLCOMPLCOND_BUSY
Completion of the call under busy conditions.

LINECALLCOMPLCOND_NOANSWER
Completion of the call under ringback no answer conditions.

Not extensible. All 32 bits are reserved.

LINECALLCOMPLMODE_ Constants

[New - Windows 95]

The LINECALLCOMPLMODE_ bit-flag constants describe different ways in which a call can be completed.

LINECALLCOMPLMODE_CAMPON
Queues the call until the call can be completed.

LINECALLCOMPLMODE_CALLBACK
Requests the called station to return the call when it returns to idle.

LINECALLCOMPLMODE_INTRUDE
Adds the application to the existing call at the called station (barge in).

LINECALLCOMPLMODE_MESSAGE
Leave a short predefined message for the called station (Leave Word Calling). The message to be sent is specified separately.

Not extensible. All 32 bits are reserved.

LINECALLFEATURE_ Constants

[New - Windows 95]

The LINECALLFEATURE_ constants list the operations that can be invoked on a call using this API.

Not extensible. All 32 bits are reserved.

These constants are used both in **LINEADDRESSCAPS** (returned by **TSPI_lineGetAddressCaps**) and in **LINECALLSTATUS** (returned by **TSPI_lineGetCallStatus**). **LINEADDRESSCAPS** reports the general availability of the call features on the specified address. An application can use this information when it initializes in order to determine what it may be able to do later when calls actuall do exist. **LINECALLSTATUS** reports for the specified call which call features can actually be invoked while the call is in the current call state. The latter takes call privileges into account. An application makes this determination dynamically, after call state changes.

LINECALLINFOSTATE_ Constants

[New - Windows 95]

The LINECALLINFOSTATE_ bit-flag constants describe various call information items about which an application may be notified in the **LINE_CALLINFO** message.

LINECALLINFOSTATE_OTHER
Call information-status items other than those listed below have changed.

LINECALLINFOSTATE_DEVSPECIFIC
The device-specific field of the call-information record.

LINECALLINFOSTATE_BEARERMODE
The bearer mode field of the call-information record.

LINECALLINFOSTATE_RATE
The rate field of the call-information record.

LINECALLINFOSTATE_MEDIAMODE
The media mode field of the call-information record.

LINECALLINFOSTATE_APPSPECIFIC
The application-specific field of the call-information record.

LINECALLINFOSTATE_CALLID
The call ID field of the call-information record.

LINECALLINFOSTATE_RELATEDCALLID
The related call ID field of the call-information record.

LINECALLINFOSTATE_ORIGIN
The origin field of the call-information record.

LINECALLINFOSTATE_REASON
The reason field of the call-information record.

LINECALLINFOSTATE_COMPLETIONID
The completion ID field of the call-information record.

LINECALLINFOSTATE_NUMOWNERINCR
The number of owner field in the call-information record was increased.

LINECALLINFOSTATE_NUMOWNERDECR
The number of owner field in the call-information record was decreased.

LINECALLINFOSTATE_NUMMONITORS
The number of monitors field in the call-information record has changed.

LINECALLINFOSTATE_TRUNK
The trunk field of the call-information record.

LINECALLINFOSTATE_CALLERID
One of the callerID-related fields of the call-information record.

LINECALLINFOSTATE_CALLEDID
One of the calledID-related fields of the call-information record.

LINECALLINFOSTATE_CONNECTEDID
One of the cconnectedID-related fields of the call-information record.

LINECALLINFOSTATE_REDIRECTIONID
One of the redirectionID-related fields of the call-information record.

LINECALLINFOSTATE_REDIRECTINGID
One of the redirectingID-related fields of the call-information record.

LINECALLINFOSTATE_DISPLAY
The display field of the call-information record.

LINECALLINFOSTATE_USERUSERINFO
The user-to-user information of the call-information record.

LINECALLINFOSTATE_HIGHLEVELCOMP

The high level compatibility field of the call-information record.

LINECALLINFOSTATE_LOWLEVELCOMP

The low level compatibility field of the call-information record.

LINECALLINFOSTATE_CHARGINGINFO

The charging information of the call-information record.

LINECALLINFOSTATE_TERMINAL

The terminal mode information of the call-information record.

LINECALLINFOSTATE_DIALPARAMS

The dial parameters of the call-information record.

LINECALLINFOSTATE_MONITORMODES

One or more of the digit, tone, or media monitoring fields in the call-information record.

Not extensible. All 32 bits are reserved.

These constants describe portions of the **LINECALLINFO** data structure. When changes occur in this data structure, a **LINE_CALLINFO** message is sent to the application. The parameters to this message are a handle to the call and an indication of the information item that has changed.

The **LINEADDRESSCAPS** data structure also indicates which of these call information elements are every valid for calls on the address.

LINECALLORIGIN_ Constants

[New - Windows 95]

The LINECALLORIGIN_ constants describe the origin of a call.

LINECALLORIGIN_OUTBOUND
: The call originated from this station as an outbound call.

LINECALLORIGIN_INTERNAL
: The call originated as an inbound call at a station internal to the same switching environment.

LINECALLORIGIN_EXTERNAL
: The call originated as an inbound call on an external line.

LINECALLORIGIN_UNKNOWN
: The call is an inbound call, but its origin is currently unknown but may become known later.

LINECALLORIGIN_UNAVAIL
: The call is an inbound call, and its origin is not available and will never become known for this call.

LINECALLORIGIN_CONFERENCE
: The call handle is for a conference call, that is, the application's connection to the conference bridge in the switch.

Not extensible. All 32 bits are reserved.

The origin of a call is stored in the call's **LINECALLINFO** structure.

LINECALLPARAMFLAGS_ Constants

[New - Windows 95]

The LINECALLPARAMFLAGS_ constants describe various status flags about a call.

LINECALLPARAMFLAGS_SECURE
The call should be set up as secure.

LINECALLPARAMFLAGS_IDLE
The call should be originated on an idle call appearance, and not join a call in progress.

LINECALLPARAMFLAGS_BLOCKID
The originator identity should be concealed (block caller ID).

LINECALLPARAMFLAGS_ORIGOFFHOOK
The originator's phone should be automatically taken off hook.

LINECALLPARAMFLAGS_DESTOFFHOOK
The called party's phone should be automatically be taken off hook.

Not extensible. All 32 bits are reserved.

LINECALLPARTYID_ Constants

[New - Windows 95]

The LINECALLPARTYID_ bit-flag constants describe the nature of the information available about the parties involved in a call.

LINECALLPARTYID_BLOCKED
: Party ID information is not available because it has been blocked by the remote party.

LINECALLPARTYID_OUTOFAREA
: Caller ID information for the call is not available since it is not propagated all the way by the network.

LINECALLPARTYID_NAME
: Party ID information consists of the party's name (as, for example, from a directory kept inside the switch).

LINECALLPARTYID_ADDRESS
: Party ID information consists of the party's address in dialable address format.

LINECALLPARTYID_PARTIAL
: Party ID information is valid, but is limited to partial information only.

LINECALLPARTYID_UNKNOWN
: Party ID information is currently unknown but may become known later.

LINECALLPARTYID_UNAVAIL
: Party ID information is not available and will not become later. Information may be unavailable for unspecified reasons. For example, the information was not delivered by the network, it was ignored by the service provider, and so forth.

Not extensible. All 32 bits are reserved.

LINECALLPARTYID describes for each of the possible parties involved in a call how the party ID information is formatted. This information is supplied in the **LINECALLINFO** data structure.

LINECALLREASON_ Constants

[New - Windows 95]

The LINECALLREASON_ bit-flag constants describe the reason for a call.

LINECALLREASON_DIRECT
: This is a direct inbound or outbound call.

LINECALLREASON_FWDBUSY
: This call was forwarded from another extension that was busy at the time of the call.

LINECALLREASON_FWDNOANSWER
: The call was forwarded from another extension that didn't answer the call after some number of rings.

LINECALLREASON_FWDUNCOND
: The call was forwarded unconditionally from another number.

LINECALLREASON_PICKUP
: The call was picked up from another extension.

LINECALLREASON_UNPARK
: The call was retrieved as a parked call.

LINECALLREASON_REDIRECT
: The call was redirected to this station.

LINECALLREASON_CALLCOMPLETION
: The call was the result of a call completion request.

LINECALLREASON_TRANSFER
: The call has been transferred from another number.

LINECALLREASON_REMINDER
: The call is a reminder (or "recall") that the user has a call parked or on hold for (potentially) a long time.

LINECALLREASON_UNKNOWN
: The reason for the call is currently unknown but may become known later.

LINECALLREASON_UNAVAIL
: The reason for the call is unavailable and will not become known later.

Not extensible. All 32 bits are reserved.

The LINECALLREASON_ constants are used in the **dwReason** field of the **LINECALLINFO** data structure.

LINECALLSELECT_ Constants

[New - Windows 95]

The LINECALLSELECT_ bit-flag constants describe which calls are to be selected.

LINECALLSELECT_LINE
Selects calls on this specified line device.

LINECALLSELECT_ADDRESS
Selects call on the specified address.

LINECALLSELECT_CALL
Selects related calls to the specified call. Examples are the parties in a conference call.

Not extensible. All 32 bits are reserved.

This constant is used in the functions listed below to specify a selection (scope) of the calls that are requested.

LINECALLSTATE_ Constants

[New - Windows 95]

The LINECALLSTATE_ bit-flag constants describe the call states a call can be in.

LINECALLSTATE_IDLE
: The call is idle—no call actually exists.

LINECALLSTATE_OFFERING
: The call is being offered to the station, signaling the arrival of a new call. In some environments, a call in the *offering* state does not automatically alert the user; alerting is done by the switch instructing the line to ring, it does not affect any call states.

LINECALLSTATE_ACCEPTED
: The call was offering and has been accepted. This indicates to other (monitoring) applications that the current owner application has claimed responsibility for answering the call. In ISDN, this also initiates alerting to both parties.

LINECALLSTATE_DIALTONE
: The call is receiving a dial tone from the switch, which means that the switch is ready to receive a dialed number.

LINECALLSTATE_DIALING
: Destination address information (a phone number) is being sent to the switch over the call. Note that the operation **TSPI_lineGenerateDigits** does not place the line into the *dialing* state.

LINECALLSTATE_RINGBACK
: The call is receiving ringback from the called address. Ringback indicates that the other station has been reached and is being alerted.

LINECALLSTATE_BUSY
: The call is receiving a busy tone. A busy tone indicates that the call cannot be completed because either a circuit (trunk) or the remote party's station are in use.

LINECALLSTATE_SPECIALINFO
: Special information is sent by the network. Special information is typically sent when the destination cannot be reached.

LINECALLSTATE_CONNECTED
: The call has been established and the connection is made. Information is able to flow over the call between the originating address and the destination address.

LINECALLSTATE_PROCEEDING
: Dialing has completed and the call is proceeding through the switch or telephone network.

LINECALLSTATE_ONHOLD
: The call is on hold by the switch.

LINECALLSTATE_CONFERENCED
: The call is currently a member of a multi-party conference call.

LINECALLSTATE_ONHOLDPENDCONF
: The call is currently on hold while it is being added to a conference.

LINECALLSTATE_ONHOLDPENDTRANSFER
: The call is currently on hold awaiting transfer to another number.

LINECALLSTATE_DISCONNECTED
: The remote party has disconnected from the call.

LINECALLSTATE_UNKNOWN
: The state of the call is not known. This may be due to limitations of the call-progress detection implementation.

The high-order 8 bits can define a device-specific substate of any of the predefined states, provided that one of the LINECALLSTATE_ bits defined above is also set. The low-order 24 bits are reserved for predefined states.

The LINECALLSTATE_ constants are used as parameters by the **LINE_CALLSTATE** message sent to the application, the message carries the new call state that the call transitioned to. These constants are

also used in fields of the **LINECALLSTATUS** structure, returned by the **TSPI_lineGetCallStatus** function.

LINEDEVCAPFLAGS_ Constants

[New - Windows 95]

The LINEDEVCAPFLAGS_ bit-flag constants are a collection of BOOLEANs describing various line device capabilities.

LINEDEVCAPFLAGS_CROSSADDRCONF
Specifies whether calls on different addresses on this line can be conferenced.

LINEDEVCAPFLAGS_HIGHLEVCOMP
Specifies whether high-level compatibility information elements are supported on this line.

LINEDEVCAPFLAGS_LOWLEVCOMP
Specifies whether low-level compatibility information elements are supported on this line.

LINEDEVCAPFLAGS_MEDIACONTROL
Specifies whether media control operations are available for calls at this line.

LINEDEVCAPFLAGS_MULTIPLEADDR
Specifies whether **TSPI_lineMakeCall** or **TSPI_lineDial** are able to deal with multiple addresses at once (as for inverse multiplexing).

LINEDEVCAPFLAGS_CLOSEDROP
Specifies what happens when an open line is closed while the application has calls active on the line. If TRUE, a **TSPI_lineClose** will drop (that is, clear) all calls on the line if the application is the sole owner of those calls. Knowing the setting of this flag ahead of time makes it possible for the application to display an OK/Cancel dialog box for the user, warning that the active call will be lost. If CLOSEDROP is FALSE, a **TSPI_lineClose** will not automatically drop any calls still active on the line if the service provider knows that some other device can keep the call alive. For example, if an analog line has the computer and phoneset both connect directly to them (in a party-line configuration), the service provider should set the flag to FALSE, as the offhook phone will automatically keep the call active even after the computer powers down.

LINEDEVCAPFLAGS_DIALBILLING
LINEDEVCAPFLAGS_DIALQUIET
LINEDEVCAPFLAGS_DIALDIALTONE
These flags indicate whether the “$”, “@”, or “W” dialable string modifier is supported for a given line device. It is TRUE if the modifier is supported; FALSE otherwise. The “?” (prompt user to continue dialing) is never supported by a line device. These flags allow an application to determine “up front” which modifiers would result in the generation of a LINEERR_. The application has the choice of pre-scanning dialable strings for unsupported characters, or passing the “raw” string from **lineTranslateAddress** directly to the provider as part of **lineMakeCall** (or **lineDial**) and letting the function generate an error to tell it which unsupported modifier occurs first in the string.

Not extensible. All 32 bits are reserved.

LINEDEVSTATE_ Constants

[New - Windows 95]

The LINEDEVSTATE_ bit-flag constants describe various line status events.

LINEDEVSTATE_OTHER
Device-status items other than those listed below have changed.

LINEDEVSTATE_RINGING
The switch tells the line to alert the user.

LINEDEVSTATE_CONNECTED
The line was previously disconnected and is now connected to TAPI.

LINEDEVSTATE_DISCONNECTED
This line was previously connected and is now disconnected from TAPI.

LINEDEVSTATE_MSGWAITON
The “message” waiting indicator is turned on.

LINEDEVSTATE_MSGWAITOFF
The “message waiting” indicator is turned off.

LINEDEVSTATE_INSERVICE
The line is connected to TAPI. This happens when TAPI is first activated, or when the line wire is physically plugged in and in-service at the switch while TAPI is active.

LINEDEVSTATE_OUTOFSERVICE
The line is out of service at the switch or physically disconnected. TAPI cannot be used to operate on the line device.

LINEDEVSTATE_MAINTENANCE
Mainatenance is being performed on the line at the switch. TAPI cannot be used to operate on the line device.

LINEDEVSTATE_OPEN
The line has been opened by another application.

LINEDEVSTATE_CLOSE
The line has been closed by another application.

LINEDEVSTATE_NUMCALLS
The number of calls on the line device has changed.

LINEDEVSTATE_NUMCOMPLETIONS
The number of outstanding call completions on the line device has changed.

LINEDEVSTATE_TERMINALS
The terminal settings have changed. This may happen, for example, if multiple line devices share terminals (for example, two lines sharing a phone terminal).

LINEDEVSTATE_ROAMMODE
The roam mode of the line device has changed.

LINEDEVSTATE_BATTERY
The battery level has changed significantly (cellular).

LINEDEVSTATE_SIGNAL
The signal level has changed significantly (cellular).

LINEDEVSTATE_DEVSPECIFIC
The line’s device-specific information has changed.

LINEDEVSTATE_REINIT
Items have changed in the configuration of line devices. To become aware of these changes (as for the appearance of new line devices) the application should reinitialize its use of TAPI.

LINEDEVSTATE_LOCK
This value is most often used with line devices associated with cellular phones. Many cellular phones have a security mechanism which requires the entry of a password to enable the phone to place calls. This value may be used to indicate to TAPI.DLL and its client applications that the phone

is locked and cannot place calls until the password is entered on the user interface of the phone, so that the application can present an appropriate alert to the user.

Not extensible. All 32 bits are reserved.

LINEDEVSTATUSFLAGS_ Constants

[New - Windows 95]

The LINEDEVSTATUSFLAGS_ bit-flag constants describe a collection of BOOLEAN line device status items.

LINEDEVSTATUSFLAGS_CONNECTED

Specifies whether the line is connected to TAPI. If TRUE, the line is connected, TAPI is able to operate on the line device. If FALSE, the line is disconnected, and the application is unable to control the line device through TAPI.

LINEDEVSTATUSFLAGS_MSGWAIT

This field indicates whether the line has a message waiting. If TRUE, a message is waiting; if FALSE, no message is waiting.

LINEDEVSTATUSFLAGS_INSERVICE

This field indicates whether the line is in service. If TRUE, the line is in service; if FALSE, the line is out of service.

LINEDEVSTATUSFLAGS_LOCKED

The bit represented by this value is most often used with line devices associated with cellular phones. Many cellular phones have a security mechanism which requires the entry of a password to enable the phone to place calls. The bit represented by this value may be used to indicate to the TAPI.DLL and its client applications that the phone is locked and cannot place calls until the password is entered on the user interface of the phone, so that the application can present an appropriate alert to the user.

Not extensible. All 32 bits are reserved.

LINEDIALTONEMODE_ Constants

[New - Windows 95]

The LINEDIALTONEMODE_ bit-flag constants describe different types of dialtones. A special dialtone typically carries a special meaning (as with message waiting).

LINEDIALTONEMODE_NORMAL
: This is a “normal” dialtone, which is typically a continuous tone.

LINEDIALTONEMODE_SPECIAL
: A special dialtone indicating a certain condition (known by the switch or network) is currently in effect. Special dialtones typically use an interrupted tone. As with a normal dial tone, this indicates that the switch is ready to receive the number to be dialed.

LINEDIALTONEMODE_INTERNAL
: This an internal dialtone, as within a PBX.

LINEDIALTONEMODE_EXTERNAL
: This is an external (public network) dialtone.

LINEDIALTONEMODE_UNKNOWN
: The dialtone mode is currently known, but may become known later.

LINEDIALTONEMODE_UNAVAIL
: The dialtone mode is unavailable and will not become known.

The high-order 16 bits can be assigned for device-specific extensions. The low-order 16 bits are reserved.

The LINEDIALTONEMODE_ constants are used in the **LINECALLSTATUS** data structure for a call in the *dialtone* state.

LINEDIGITMODE_ Constants

[New - Windows 95]

The LINEDIGITMODE_ constants describe different types of inband digit generation.

LINEDIGITMODE_PULSE
Uses rotary pulse sequences to signal digits. Valid digits are ‘0’ through ‘9’.

LINEDIGITMODE_DTMF
Uses DTMF tones to signal digits. Valid digits are ‘0’ through ‘9’, ‘*’, ‘#’, ‘A’, ‘B’, ‘C’, and ‘D’.

LINEDIGITMODE_DTMFEND
Uses DTMF tones to signal digits and detect the down edges. Valid digits are ‘0’ through ‘9’, ‘*’, ‘#’, ‘A’, ‘B’, ‘C’, and ‘D’.

Not extensible. All 32 bits are reserved.

A digit mode can be specified when generating or detecting digits. Note that pulse digits are generated by making and breaking the local loop circuit. These pulses are absorbed by the switch. The remote end merely observes this as a series of inband audio clicks. Detecting digits sent as pulses must therefore be able to detect these sequences of 1 to 10 audible clicks.

LINEDISCONNECTMODE_ Constants

[New - Windows 95]

The LINEDISCONNECTMODE_ bit-flag constants describe different reasons for a remote disconnect request. A disconnect mode is available as a call status to the application after the call state transitions to *disconnected*.

LINEDISCONNECTMODE_NORMAL
This is a “normal” disconnect request by the remote party; the call was terminated normally.

LINEDISCONNECTMODE_UNKNOWN
The reason for the disconnect request is unknown, but may become known later.

LINEDISCONNECTMODE_REJECT
The remote user has rejected the call.

LINEDISCONNECTMODE_PICKUP
The call was picked up from elsewhere.

LINEDISCONNECTMODE_FORWARDED
The call was forwarded by the switch.

LINEDISCONNECTMODE_BUSY
The remote user’s station is busy.

LINEDISCONNECTMODE_NOANSWER
The remote user’s station does not answer.

LINEDISCONNECTMODE_BADADDRESS
The destination address in invalid.

LINEDISCONNECTMODE_UNREACHABLE
The remote user could not be reached.

LINEDISCONNECTMODE_CONGESTION
The network is congested.

LINEDISCONNECTMODE_INCOMPATIBLE
The remote user’s station equipment is incompatible for the type of call requested.

LINEDISCONNECTMODE_UNAVAIL
The reason for the disconnect is unavailable and will not become known later.

The high-order 16 bits can be assigned for device-specific extensions. The low-order 16 bits are reserved.

A remote disconnect request for a given call results in the call state transitioning to the *disconnected* state and a **LINE_CALLSTATE** messages is sent to the application. The LINEDISCONNECTMODE_ constants provide details about the remote disconnect request. It is available in the call’s **LINECALLSTATUS** structure when the call is in the *disconnected* state. While a call is in this state, the application is still allowed to query the calls information and status. For example, user-to-user information received as part of the remote disconnect is available then. The application can clear a *disconnected* call by dropping the call.

LINEERR_ Constants

[New - Windows 95]

This is the list of error codes that the implementation may return when invoking operations on lines, addresses, or calls. Consult the individual function descriptions to determine which of these error codes a particular function may return.

LINEERR_ADDRESSBLOCKED

The specified address is blocked from being dialed on the specified call.

LINEERR_ALLOCATED

The line cannot be opened due to a “persistent” condition, such as that of a serial port being exclusively opened by another process.

LINEERR_BEARERMODEUNAVAIL

The bearer mode field in **LINECALLPARAMS** is invalid, the bearer mode specified in **LINECALLPARAMS** is not available, or the call’s bearer mode cannot be changed to the specified bearer mode.

LINEERR_CALLUNAVAIL

All call appearances on the specified address are currently in use or allocated.

LINEERR_COMPLETIONOVERRUN

The maximum number of outstanding call completions has been exceeded.

LINEERR_CONFERENCEFULL

The maximum number of parties for a conference has been reached or the requested number of parties cannot be satisfied.

LINEERR_DIALBILLING

The dialable address parameter contains dialing control characters that are not processed by the service provider.

LINEERR_DIALDIALTONE

The dialable address parameter contains dialing control characters that are not processed by the service provider.

LINEERR_DIALPROMPT

The dialable address parameter contains dialing control characters that are not processed by the service provider.

LINEERR_DIALQUIET

The dialable address parameter contains dialing control characters that are not processed by the service provider.

LINEERR_INCOMPATIBLEAPIVERSION

The passed TSPI version or version range did not match an interface version definition supported by the service provider.

LINEERR_INCOMPATIBLEEXTVERSION

The passed extension version range did not contain an interface definition version supported by the service provider, or the passed TSPI version did not match an extension version definition supported by the service provider.

LINEERR_INUSE

The line device is in use and cannot currently be configured, allow a party to be added, allow a call to be answered, or allow a call to be placed.

LINEERR_INVALADDRESS

The specified address is invalid, is not assigned to the specified line, contains invalid characters or digits, or is not valid for redirection. This error is also returned if the destination address contains dialing control characters (W, @, $, ?) that are not supported by the server provider.

LINEERR_INVALADDRESSID

The specified address ID, such as the *dwAddressID* parameter or in the **LINECALLPARAMS** structure, is invalid or is out of range.

LINEERR_INVALADDRESSMODE

The address mode specified in **LINECALLPARAMS** is invalid.

LINEERR_INVALADDRESSSTATE
The specified address state is invalid.
LINEERR_INVALBEARERMODE
The bearer mode, such as specified in **LINECALLPARAMS**, is invalid.
LINEERR_INVALCALLCOMPLMODE
The specified completion is invalid.
LINEERR_INVALCALLHANDLE
One or more specified call or device handles are invalid.
LINEERR_INVALCALLPARAMS
The specified call parameters, such as in the **LINECALLPARAMS** structure, are invalid.
LINEERR_INVALCALLSELECT
The specified select parameter is invalid.
LINEERR_INVALCALLSTATE
One or more of the specified calls are not in a valid state for the requested operation.
LINEERR_INVALCALLSTATELIST
The specified call state list is invalid.
LINEERR_INVALCOMPLETIONID
The completion ID is invalid.
LINEERR_INVALCONFCALLHANDLE
The specified call handle for the conference call is invalid or is not a handle for a conference call.
LINEERR_INVALCOUNTRYCODE
One or more of the specified country codes, such as in the **LINEFORWARDLIST** structure, are invalid.
LINEERR_INVALDEVICECLASS
The line device has no associated device for the given device class, or the specified line does not support the indicated device class.
LINEERR_INVALDIGITLIST
The specified digit list is invalid.
LINEERR_INVALDIGITMODE
The specified digit mode is invalid or not available.
LINEERR_INVALDIGITS
The specified termination digits are not valid.
LINEERR_INVALFEATURE
The *dwFeature* parameter is invalid.
LINEERR_INVALGROUPID
The specified group ID is invalid.
LINEERR_INVALLINEHANDLE
The specified call, device, line device, or line handle is invalid.
LINEERR_INVALLINESTATE
The line is currently not in a state in which this operation can be performed.
LINEERR_INVALMEDIALIST
The specified media list is invalid.
LINEERR_INVALMEDIAMODE
One or more media modes specified as a parameter, in the media mode field in the **LINECALLPARAMS** structure, or in a list is invalid or not supported by the the service provider.
LINEERR_INVALMESSAGEID
The number given in *dwMessageID* is outside the range specified by the **dwNumCompletionMessages** field in the **LINEADDRESSCAPS** structure.
LINEERR_INVALPARAM
A parameter or structure pointed to by a parameter contains invalid information.

LINEERR_INVALPARKMODE

The specified park mode is invalid.

LINEERR_INVALPOINTER

One or more of the specified pointer parameters are invalid.

LINEERR_INVALPOINTER

The *lpDialParams* pointer is non-NULL and invalid.

LINEERR_INVALPOINTER

This error return value is returned if *lpToneList* has a non-NULL value that is invalid.

LINEERR_INVALRATE

One or more rates, such as specified in the **LINECALLPARAMS** structure, are invalid.

LINEERR_INVALTERMINALID

The specified terminal mode parameter is invalid.

LINEERR_INVALTERMINALMODE

The specified terminal modes parameter is invalid.

LINEERR_INVALTIMEOUT

The values of either or both of the parameters *dwFirstDigitTimeout* or *dwInterDigitTimeout* fall outside the valid range specified by **LINEDEVCAPS**, or timeouts are not supported.

LINEERR_INVALTONE

The specified tone structure or custom tone does not represent or describe a valid tone, or is made up of too many frequencies.

LINEERR_INVALTONELIST

The specified tone list is invalid.

LINEERR_INVALTONEMODE

The specified tone mode parameter is invalid.

LINEERR_NODEVICE

The line device has no associated device for the given device class.

LINEERR_NODRIVER

This error may be returned from a function when the driver finds that one of its components is corrupt in a way that was not detected at initialization time. For example, if a service provider consists of multiple components such as a Virtual Device Driver (VxD) in addition to the usual DLL (exporting the TSPI interface and having the ".TSP" extension), LINEERR_NODRIVER could be returned to indicate that the VxD component was missing. The user should be advised to use the Telephony Control Panel to correct the problem.

LINEERR_NOMEM

Unable to allocate or lock memory.

LINEERR_OPERATIONFAILED

The operation failed for unspecified reasons.

LINEERR_OPERATIONUNAVAIL

The operation is not available.

LINEERR_RATEUNAVAIL

The service provider currently does not have enough bandwidth available for the rate specified in **LINECALLPARAMS**.

LINEERR_RATEUNAVAIL

The service provider does currently not have enough bandwidth available for the specified rate, such as given in **LINECALLPARAMS.**

LINEERR_RESOURCEUNAVAIL

The operation cannot be completed because of resource overcommitment.

LINEERR_RESOURCEUNAVAIL

The operation cannot be completed because of resource overcommitment. This return value may also be returned if too many terminals are set, due either to hardware limitations or to service provider/device driver limitations.

LINEERR_STRUCTURETOOSMALL

The **dwTotalSize** member of a structure, such as a **VARSTRING** or **LINEFORWARDLIST** structure, does not specify enough memory to contain the fixed portion of the structure. The **dwNeededSize** field has been set to the amount required.

LINEERR_USERUSERINFOTOOBIG

The string containing user-to-user information is either too long or exceeds the maximum number of bytes specified in one of these members of the **LINEDEVCAPS** structure: **dwUUIAcceptSize**, **dwUUIAnswerSize**, **dwUUIDropSize**, **dwUUIMakeCallSize**, or **dwUUISendUserUserInfoSize**.

The values 0xC0000000 through 0xFFFFFFFF are available for device-specific extensions. The values 0x80000000 through 0xBFFFFFFF are reserved, while 0x00000000 through 0x7FFFFFFF are used as request IDs.

LINEFEATURE_ Constants

[New - Windows 95]

The LINEFEATURE_ constants list the operations that can be invoked on a line using this API.

LINEFEATURE_DEVSPECIFIC
Device-specific operations can be used on the line.

LINEFEATURE_DEVSPECIFICFEAT
Device-specific features can be used on the line.

LINEFEATURE_FORWARD
Forwarding of all addresses can be used on the line.

LINEFEATURE_MAKECALL
An outbound call can be placed on this line using an unspecified address.

LINEFEATURE_SETMEDIACONTROL
Media control can be set on this line.

LINEFEATURE_SETTERMINAL
Terminal modes for this line can be set.

Not extensible. All 32 bits are reserved.

This constant is used both in **LINEDEVCAPS** (returned by **TSPI_lineGetDevCaps**) and in **LINEDEVSTATUS** (returned by **TSPI_lineGetLineDevStatus**). **LINEDEVCAPS** reports the availability of the line features by the service provider (mainly the switch) for a given line. An application makes this determination when it initializes. **LINEDEVSTATUS** reports for a given line which line features can actually be invoked while the line is in the current state. An application would make this determination dynamically, after line state changes, typically caused by address or call-related activities on the line.

LINEFORWARDMODE_ Constants

[New - Windows 95]

The LINEFORWARDMODE_ bit-flag constants describe the conditions under which calls to an address can be forwarded.

LINEFORWARDMODE_UNCOND
: Forward all calls unconditionally irrespective of their origin. Use this value when unconditional forwarding for internal and external calls cannot be controlled separately. Unconditional forwarding overrides forwarding on busy and/or no answer conditions.

LINEFORWARDMODE_UNCONDINTERNAL
: Forward all internal calls unconditionally. Use this value when unconditional forwarding for internal and external calls can be controlled separately.

LINEFORWARDMODE_UNCONDEXTERNAL
: Forward all external calls unconditionally. Use this value when unconditional forwarding for internal and external calls can be controlled separately.

LINEFORWARDMODE_UNCONDSPECIFIC
: Forward all calls that originated at a specified address unconditionally (selective call forwarding).

LINEFORWARDMODE_BUSY
: Forward all calls on busy irrespective of their origin. Use this value when forwarding for internal and external calls on busy and no answer cannot be controlled separately.

LINEFORWARDMODE_BUSYINTERNAL
: Forward all internal calls on busy. Use this value when forwarding for internal and external calls on busy and no answer can be controlled separately.

LINEFORWARDMODE_BUSYEXTERNAL
: Forward all external calls on busy. Use this value when forwarding for internal and external calls on busy and no answer can be controlled separately.

LINEFORWARDMODE_BUSYSPECIFIC
: Forward all calls that originated at a specified address on busy (selective call forwarding).

LINEFORWARDMODE_NOANSW
: Forward all calls on no answer irrespective of their origin. Use this value when call forwarding for internal and external calls on no answer cannot be controlled separately.

LINEFORWARDMODE_NOANSWINTERNAL
: Forward all internal calls on no answer. Use this value when forwarding for internal and external calls on no answer can be controlled separately.

LINEFORWARDMODE_NOANSWEXTERNAL
: Forward all external calls on no answer. Use this value when forwarding for internal and external calls on no answer can be controlled separately.

LINEFORWARDMODE_NOANSWSPECIFIC
: Forward all calls that originated at a specified address on no answer (selective call forwarding).

LINEFORWARDMODE_BUSYNA
: Forward all calls on busy/no answer irrespective of their origin. Use this value when forwarding for internal and external calls on busy and no answer cannot be controlled separately.

LINEFORWARDMODE_BUSYNAINTERNAL
: Forward all internal calls on busy/no answer. Use this value when call forwarding on busy and no answer cannot be controlled separately for internal calls.

LINEFORWARDMODE_BUSYNAEXTERNAL
: Forward all external calls on busy/no answer. Use this value when call forwarding on busy and no answer cannot be controlled separately for internal calls.

LINEFORWARDMODE_BUSYNASPECIFIC
: Forward all calls that originated at a specified address on busy/no answer (selective call forwarding).

Not extensible. All 32 bits are reserved.

The bit flags defined by LINEFORWARDMODE_ are not orthogonal. Unconditional forwarding ignores

any specific condition such as busy or no answer. If unconditional forwarding is not in effect, forwarding on busy and no answer can either be controlled separately or not. If controlled separately, the BUSY and NOANSW flags can be used separately. If not controlled separately, the flag BUSYNA must be used. Similarly, if forwarding of internal and external calls can be controlled separately, INTERNAL and EXTERNAL flags can be used separately, otherwise the combination is used.

Address capabilities indicate for each address assigned to a line which forwarding mode(s) are available. An application can use **TSPI_lineForward** to set forwarding conditions at the switch.

LINEGATHERTERM_ Constants

[New - Windows 95]

The LINEGATHERTERM_ bit-flag constants describe the conditions under which buffered digit gathering is terminated.

LINEGATHERTERM_BUFFERFULL
: The requested number of digits has been gathered. The buffer is full.

LINEGATHERTERM_TERMDIGIT
: One of the termination digits matched a received digit. The matched termination digit is the last digit in the buffer.

LINEGATHERTERM_FIRSTTIMEOUT
: The first digit timeout expired. The buffer contains no digits.

LINEGATHERTERM_INTERTIMEOUT
: The inter digit timeout expired. The buffer contains at least one digit.

LINEGATHERTERM_CANCEL
: The request was canceled by this application, by another application, or because the call terminated.

Not extensible. All 32 bits are reserved.

LINEGENERATETERM_ Constants

[New - Windows 95]

The LINEGENERATETERM_ bit-flag constants describe the conditions under which digit or tone generation is terminated.

LINEGENERATETERM_DONE

The requested number of digits have been generated, or requested tones have been generated for the requested duration.

LINEGENERATETERM_CANCEL

The digit or tone generation request was canceled by this application, by another application, or because the call terminated. LINEGENERATETERM_CANCEL can also be returned when digit or tone generation cannot be completed due to an internal failure of the service provider.

Not extensible. All 32 bits are reserved.

LINEMEDIACONTROL_ Constants

[New - Windows 95]

The LINEMEDIACONTROL_ bit-flag constants describe a set of generic operations on media streams. The interpretations are determined by the media stream. The line device must have the media control capability in order for any media control operations to be effective.

LINEMEDIACONTROL_NONE
No change is to be made to the media stream.

LINEMEDIACONTROL_START
Start the media stream.

LINEMEDIACONTROL_RESET
Reset the media stream. Provide the effect of an end-of-input. All buffers are released.

LINEMEDIACONTROL_PAUSE
Temporarily pause the media stream.

LINEMEDIACONTROL_RESUME
Resume a paused media stream.

LINEMEDIACONTROL_RATEUP
The speed of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_RATEDOWN
The speed of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_RATENORMAL
The speed of the media stream is returned to normal.

LINEMEDIACONTROL_VOLUMEUP
The amplitude of the media stream is increased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMEDOWN
The amplitude of the media stream is decreased by some stream-defined quantity.

LINEMEDIACONTROL_VOLUMENORMAL
The amplitude of the media stream is returned to normal.

The high-order 16 bits can be assigned for device-specific extensions. The low-order 16 bits are reserved.

Media control is provided as a way of improving the performance of actions on media streams on calls in response to telephony related events. The application should normally manage a media stream through the media-specific API; the media control functionality provided here is not intended as a replacemenet for the native media APIs.

Media control actions can be associated with the detection of digits, the detection of tones, the transition into a call state, and the detection of a media mode. Consult a line’s device capabilities to determine whether media control is available on a line.

LINEMEDIAMODE_ Constants

[New - Windows 95]

The LINEMEDIAMODE_ constants describe the common media modes (the data types of a media stream) on calls. They are used in a system of priority ordering when presenting a new incoming call of UNKNOWN media mode with multiple specific media mode flags set. The priority determines which media-handling application the call should be presented to first. (Extension media modes are considered to all have a lower priority order than the common media modes.)

LINEMEDIAMODE_UNKNOWN
: A media stream exists but its mode is not currently known and may become known later. This corresponds to a call with an unclassified media type. In typical analog telephony environments, an inbound call's media mode may be unknown until after the call has been answered and the media stream filtered to make a determination.

LINEMEDIAMODE_INTERACTIVEVOICE
: The presence of voice energy on the call and the call is treated as an interactive call with humans on both ends.

LINEMEDIAMODE_AUTOMATEDVOICE
: The presence of voice energy on the call and the voice is locally handled by an automated application.

LINEMEDIAMODE_DATAMODEM
: A data modem session on the call.

LINEMEDIAMODE_G3FAX
: A group 3 fax is being sent or received over the call.

LINEMEDIAMODE_TDD
: A TDD (Telephony Devices for the Deaf) session on the call.

LINEMEDIAMODE_G4FAX
: A group 4 fax is being sent or received over the call.

LINEMEDIAMODE_DIGITALDATA
: Digital data is being sent or received over the call.

LINEMEDIAMODE_TELETEX
: A teletex session on the call. Teletex is one of the telematic services.

LINEMEDIAMODE_VIDEOTEX
: A videotex session on the call. Videotex is one the telematic services.

LINEMEDIAMODE_TELEX
: A telex session on the call. Telex is one the telematic services.

LINEMEDIAMODE_MIXED
: A mixed session on the call. Mixed is one the ISDN telematic services.

LINEMEDIAMODE_ADSI
: An ADSI (Analog Display Services Interface) session on the call.

The high-order 8 bits can be assigned for device-specific extensions. The low-order 24 bits are reserved.

Note that bearer mode and media mode are different notions. The bearer mode of a call is an indication of the quality of the telephone connection as provided primarily by the network. The media mode of a call is an indication of the type of information stream that is exchanged over that call. Group 3 fax or data modem are media modes that use a call with a 3.1kHz voice bearer mode.

LINEPARKMODE_ Constants

[New - Windows 95]

The LINEPARKMODE_ bit-flag constants describe different ways of parking calls.

LINEPARKMODE_DIRECTED
: Specifies directed call park. The address where the call is to be parked must be supplied to the switch.

LINEPARKMODE_NONDIRECTED
: Specifies nondirected call park. The address where the call is parked is selected by the switch and provided by the switch to the application.

Not extensible. All 32 bits are reserved.

The LINEPARKMODE_ constants are used when parking a call. Consult a line's address device capabilities to find out which park mode is available.

LINEREMOVEFROMCONF_ Constants

[New - Windows 95]

The LINEREMOVEFROMCONF_ scalar constants describe how parties participating in a conference call can be removed from a conference call.

LINEREMOVEFROMCONF_NONE
Parties cannot be removed from the conference call.

LINEREMOVEFROMCONF_LAST
Only the most recently added party can be removed from the conference call

LINEREMOVEFROMCONF_ANY
Any participating party can be removed from the conference call.

Not extensible. All 32 bits are reserved.

LINEROAMMODE_ Constants

[New - Windows 95]

The LINEROAMMODE_ bit-flag constants describe the roaming status of a line device.

LINEROAMMODE_UNKNOWN
The roam mode is currently unknown, but may become known later.

LINEROAMMODE_UNAVAIL
The roam mode is unavailable and will not be known.

LINEROAMMODE_HOME
The line is connected to the home network node.

LINEROAMMODE_ROAMA
The line is connected to the Roam-A carrier and calls are charged accordingly.

LINEROAMMODE_ROAMB
The line is connected to the Roam-B carrier and calls are charged accordingly.

Not extensible. All 32 bits are reserved.

LINESPECIALINFO_ Constants

[New - Windows 95]

The LINESPECIALINFO_ bit-flag constants describes special information signals that the network may use to report various reporting and network observation operations. They are special coded tone sequences transmitted at the beginning of network advisory recorded announcements.

LINESPECIALINFO_NOCIRCUIT

This special information tone preceeds a no circuit or emergency announcement (trunk blockage category).

LINESPECIALINFO_CUSTIRREG

This special information tone preceeds a vacant number, AIS, Centrex number change and non-working station, access code not dialed or dialed in error, and manual intercept operator message (customer irregularity category). LINESPECIALINFO_CUSTIRREG is also reported when billing information is rejected and when the dialed address is blocked at the switch.

LINESPECIALINFO_REORDER

This special information tone preceeds a reorder announcement (equipment irregularity category). LINESPECIALINFO_REORDER is also reported when the telephone is kept off-hook too long.

LINESPECIALINFO_UNKNOWN

Specifics about the special information tone are currently unknown but may become known later.

LINESPECIALINFO_UNAVAIL

Specifics about the special information tone are unavailable, and will not become known.

The high-order 16 bits can be assigned for device-specific extensions. The low-order 16 bits are reserved.

Special information tones are defined for advisory messages and are not usually used for billing or supervisory purpose.

LINETERMDEV_ Constants

[New - Windows 95]

The LINETERMDEV_ bit-flag constants describe different types of terminal devices.

LINETERMDEV_PHONE
The terminal is a phone set.

LINETERMDEV_HEADSET
The terminal is a headset.

LINETERMDEV_SPEAKER
The terminal is an external speaker and microphone.

Not extensible. All 32 bits are reserved.

These constants are used to characterize a line's terminal device. This allows an application to determine the nature of a terminal device.

LINETERMMODE_ Constants

[New - Windows 95]

The LINETERMMODE_ bit-flag constants describe different types of events on a phone line that can be routed to a terminal device.

LINETERMMODE_BUTTONS
: These are button press events sent from the terminal to the line.

LINETERMMODE_LAMPS
: These are lamp events sent from the line to the terminal.

LINETERMMODE_DISPLAY
: This is display information sent from the line to the terminal.

LINETERMMODE_RINGER
: This is ringer control information sent from the switch to the terminal.

LINETERMMODE_HOOKSWITCH
: These are hookswitch events sent from the terminal to the line.

LINETERMMODE_MEDIATOLINE
: This is the unidirectional media stream from the terminal to the line associated with a call on the line. Use this value when routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIAFROMLINE
: This is the unidirectional media stream from the line to the terminal associated with a call on the line. Use this value when routing of both unidirectional channels of a call's media stream can be controlled independently.

LINETERMMODE_MEDIABIDIRECT
: This is the bidirectional media stream associated with a call on the line and the terminal. Use this value when routing of both unidirectional channels of a call's media stream cannot be controlled independently.

Not extensible. All 32 bits are reserved.

These constants describe the classes of control and information streams that can be routed directly between a line device and a terminal device (such as a phone set).

LINETERMSHARING_ Constants

[New - Windows 95]

The LINETERMSHARING_ bit flag constants describe different ways in which a terminal can be shared between line devices, addresses, or calls.

LINETERMSHARING_PRIVATE
: The terminal device is private to a single line device.

LINETERMSHARING_SHAREDEXCL
: The terminal device can be used by multiple lines. The last line device to do a **TSPI_lineSetTerminal** to the terminal for a given terminal mode will have exclusive connection to the terminal for that mode.

LINETERMSHARING_SHAREDCONF
: The terminal device can be used by multiple lines. The **TSPI_lineSetTerminal** requests of the various terminals end up being “merged” or conferenced at the terminal.

Not extensible. All 32 bits are reserved.

These constants describe the classes of control and information streams that can be routed directly between a line device and a terminal device (such as a phone set).

LINETONEMODE_ Constants

[New - Windows 95]

The LINETONEMODE_ constants describe different selections used when generating line tones.

LINETONEMODE_CUSTOM
The tone is a custom tone, defined by its component frequencies, of type **LINEGENERATETONE**.

LINETONEMODE_RINGBACK
The tone is ringback tone. Exact definition is service-provider-defined.

LINETONEMODE_BUSY
The tone is a busy tone. Exact definition is service-provider-defined.

LINETONEMODE_BEEP
The tone is a beep, such as used to announce the beginning of a recording. Exact definition is service provider defined.

LINETONEMODE_BILLING
The tone is billing information tone such as a credit card prompt tone. Exact definition is service-provider-defined.

The high-order 16 bits can be assigned for device-specific extensions. The low-order 16 bits are reserved.

These constants are used to define tones to be generated inband over a call to the remote party. Note that tone detection of non-custom tones does not use these constants.

LINETRANSFERMODE_ Constants

[New - Windows 95]

The LINETRANSFERMODE_ bit-flag constants describe different ways of resolving call transfer requests.

LINETRANSFERMODE_TRANSFER

The transfer is resolved by transferring the initial call to the consultation call. Both calls will become idle to the application.

LINETRANSFERMODE_CONFERENCE

The transfer is resolved by establishing a three-way conference between the application, the party connected to the initial call and the party connected to the consultation call. A conference call is created when this option is selected.

Not extensible. All 32 bits are reserved.

Phone Device Functions

Reference

Phone Device Functions

Reference

TSPI_phoneClose

[New - Windows 95]

```
long TSPI_phoneClose(HDRVPHONE hdPhone)
```

Closes the specified open phone device after completing or aborting all outstanding asynchronous operations on the device.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_OPERATIONFAILED
PHONEERR_NOMEM	PHONEERR_OPERATIONUNAVAIL
PHONEERR_RESOURCEUNAVAIL	

hdPhone

Specifies the service provider's opaque handle to the phone to be closed. After the phone has been successfully closed, this handle is no longer valid.

The service provider has the responsibility to report completion for every asynchronous operation. If this procedure is called for a phone on which there are outstanding asynchronous operations, the operations should be reported complete with an appropriate result or error code before this procedure returns. Generally, TAPI.DLL would wait for these to complete in an orderly fashion. However, the service provider should be prepared to handle an early call to **TSPI_phoneClose** in “abort” or “emergency shutdown” situtations.

After this procedure returns the service provider must report no further events on the phone. The service provider's opaque handle for the phone becomes invalid.

The service provider must relinquish non-sharable resources it reserves while the phone is open. For example, closing a phone accessed through a comm port and modem should result in closing the comm port, making it once available for use by other applications.

This function should always succeed except in extraordinary circumstances. Most callers will probably ignore the return code since they will be unable to compensate for any error that occurs. The specified return values are more advisory for development diagnostic purposes than anything else.

TSPI_phoneConfigDialog

[New - Windows 95]

```
LONG TSPI_phoneConfigDialog(DWORD dwDeviceID, HWND hwndOwner, LPCSTR
lpszDeviceClass)
```

Causes the provider of the specified phone device to display a modal dialog as a child window of *hwndOwner* to allow the user to configure parameters related to the phone device.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_BADDEVICEID	PHONEERR_NOMEM
PHONEERR_INUSE	PHONEERR_OPERATIONFAILED
PHONEERR_INVALPARAM	PHONEERR_OPERATIONUNAVAIL
PHONEERR_INVALDEVICECLASS	PHONEERR_RESOURCEUNAVAIL

dwDeviceID

Specifies the phone device to be configured.

hwndOwner

Specifies a handle to a parent window in which the dialog window is to be placed.

lpszDeviceClass

Specifies a far pointer to a NULL-terminated string that identifies a device class name. This device class allows the caller to select a specific subscreen of configuration information applicable to that device class. If this parameter is NULL or an empty string, the highest level configuration dialog should be selected.

TSPI_phoneConfigDialog causes the service provider to display a modal dialog as a child window of *hWndOwner* to allow the user to configure parameters related to the phone specified by *dwDeviceID*. The *lpszDeviceClass* parameter allows the application to select a specific subscreen of configuration information applicable to the device class in which the user is interested. The permitted strings are the same as for **TSPI_phoneGetID**. For example, if the phone supports the Comm API, passing “COMM” as *lpszDeviceClass* causes the provider to display the parameters related specifically to Comm (or, at least, to start at the corresponding point in a multilevel configuration dialog chain, so that the user doesn't have to search to find the desired parameters). The *szDeviceClass* parameter should be “tapi/phone”, “”, or NULL to cause the provider to display the highest level configuration for the phone.

The procedure must update the [Windows Telephony] section in WIN.INI and broadcast the **WM_WININICHANGE** message if it makes any changes to TELEPHON.INI that affect the contents of structures visible to applications (such as **PHONECAPS**), or if phone devices are created or removed.

TSPI_phoneDevSpecific

[New - Windows 95]

```
LONG TSPI_phoneDevSpecific(DRV_REQUESTID dwRequestID, HDRVPHONE hdPhone,
LPVOID lpParams, DWORD dwSize)
```

Is used as a general extension mechanism to enable a Telephony API implementation to provide features not described in the other operations. The meanings of these extensions are device specific.

- Returns *dwRequestID* or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_OPERATIONUNAVAIL
PHONEERR_INVALPOINTER	PHONEERR_NOMEM
PHONEERR_OPERATIONFAILED	PHONEERR_RESOURCEUNAVAIL

dwRequestID

Specifies the identifier of the asynchronous request.

hdPhone

Specifies the handle to the phone on which a device-specific operation is to be performed.

lpParams

Specifies a far pointer to a memory area used to hold a parameter block. Its interpretation is device specific.

dwSize

The size in bytes of the parameter block area.

Additional return values are device specific.

This operation provides a generic parameter profile. The interpretation of the parameter block is device specific. Indications and replies that are device specific should use the **PHONE_DEVSPECIFIC** message.

This function is called in direct response to an application that has called the TAPI **phoneDevSpecific** function. TAPI.DLL translates the *hPhone* parameter used at the TAPI level to the corresponding *hdPhone* parameter used at the TSPI level. The *lpParams* buffer is passed through unmodified.

A service provider can provide access to device-specific functions by defining parameters for use with this operation. Applications that want to make use of these device-specific extensions should consult the device-specific (vendor-specific) documentation that describes what extentions are defined. Note that an application that relies on these device-specific extensions will typically not be portable to work with other service provider environments.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

TSPI_phoneGetButtonInfo

[New - Windows 95]

```
LONG TSPI_phoneGetButtonInfo(HDRVPHONE hdPhone, DWORD dwButtonLampID,
LPPHONEBUTTONINFO lpButtonInfo)
```

Returns information about the specified button.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_OPERATIONFAILED
PHONEERR_INVALBUTTONLAMPID	PHONEERR_OPERATIONUNAVAIL
PHONEERR_INVALPHONESTATE	PHONEERR_RESOURCEUNAVAIL

hdPhone

Specifies the handle to the phone to be queried.

dwButtonLampID

Specifies a button on the phone device.

lpButtonInfo

Specifies a far pointer to memory into which the service provider writes a variably sized structure of type **PHONEBUTTONINFO**. This data structure describes the mode, the function, and provides additional descriptive text corresponding to the button. Prior to calling **TSPI_phoneGetButtonInfo**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI.DLL for returning information.

The service provider should fill in all the fields of the **PHONEBUTTONINFO** data structure, except for **dwTotalSize**, which is filled in by TAPI.DLL. The service provider must *not* overwrite the **dwTotalSize** field.

TSPI_phoneGetData

[New - Windows 95]

```
LONG TSPI_phoneGetData(HDRVPHONE hdPhone, DWORD dwDataID, LPVOID lpData,
DWORD dwSize)
```

Uploads the information from the specified location in the open phone device to the specified buffer.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_OPERATIONFAILED
PHONEERR_INVALPHONESTATE	PHONEERR_OPERATIONUNAVAIL
PHONEERR_INVALDATAID	PHONEERR_RESOURCEUNAVAIL

hdPhone

Specifies the handle to the phone to be queried.

dwDataID

Specifies where in the phone device the buffer is to be uploaded from.

lpData

Specifies a far pointer to the memory buffer where the data is to be uploaded to.

dwSize

Specifies the size of the data buffer in bytes.

The function uploads a maximum of *dwSize* bytes from the phone device into *lpData*. If *dwSize* is zero, nothing is copied. The size of each data area is listed in the phone's **PHONECAPS** structure.

TSPI_phoneGetDevCaps

[New - Windows 95]

```
LONG TSPI_phoneGetDevCaps(DWORD dwDeviceID, DWORD dwTSPIVersion, DWORD
dwExtVersion, LPPHONECAPS lpPhoneCaps)
```

Queries a specified phone device to determine its telephony capabilities.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INCOMPATIBLEAPIVERSION	PHONEERR_RESOURCEUNAVAIL
PHONEERR_INCOMPATIBLEEXTVERSION	PHONEERR_OPERATIONFAILED
PHONEERR_NODRIVER	PHONEERR_OPERATIONUNAVAIL
PHONEERR_NOMEM	

dwDeviceID

Specifies the phone device to be queried.

dwTSPIVersion

Specifies the negotiated TSPI version number. This value has already been negotiated for this device through the **TSPI_phoneNegotiateTSPIVersion** function.

dwExtVersion

Specifies the negotiated extension version number. This value has already been negotiated for this device through the **TSPI_phoneNegotiateExtVersion** function.

lpPhoneCaps

Specifies a far pointer to memory into which the service provider writes a variably sized structure of type **PHONECAPS**. Upon successful completion of the request, this structure is filled with phone device capabilities information. Prior to calling **TSPI_phoneGetDevCaps**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI.DLL for returning information.

The service provider should fill in all the fields of the **PHONEBUTTONINFO** data structure, except for **dwTotalSize**, which is filled in by TAPI.DLL. The service provider must *not* overwrite the **dwTotalSize** field.

If *dwExtVersion* is zero, no extension information is requested. If it is non-zero, it holds a value that has already been negotiated for this device with the function **TSPI_phoneNegotiateExtVersion**. The service provider should fill in device- and vendor-specific extended information according to the extension version specified.

TSPI_phoneGetDisplay

[New - Windows 95]

```
LONG TSPI_phoneGetDisplay(HDRVPHONE hdPhone, LPVARSTRING lpDisplay)
```

Returns the current contents of the specified phone display.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_RESOURCEUNAVAIL
PHONEERR_INVALPHONESTATE	PHONEERR_OPERATIONFAILED
PHONEERR_NOMEM	PHONEERR_OPERATIONUNAVAIL

hdPhone

Specifies the handle to the phone to be queried.

lpDisplay

Specifies a far pointer to the memory location where the display content is to be stored by the provider, of type **VARSTRING**. Prior to calling **TSPI_phoneGetDisplay**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI.DLL for returning information.

The *lpDisplay* memory area should be at least **dwNumRows** * **dwNumColumns** elements in size to receive all of the display information. The **dwNumRows** and **dwNumColumns** fields are available in the **PHONECAPS** structure returned by **TSPI_phoneGetDevCaps**.

The service provider should fill in all the fields of the **VARSTRING** data structure, except for **dwTotalSize**, which is filled in by TAPI.DLL. The service provider must *not* overwrite the **dwTotalSize** field.

TSPI_phoneGetExtensionID

[New - Windows 95]

```
LONG TSPI_phoneGetExtensionID(DWORD dwDeviceID, DWORD dwTSPIVersion,
LPPHONEEXTENSIONID lpExtensionID)
```

Retrieves the extension ID that the service provider supports for the indicated phone device.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INCOMPATIBLEAPIVERSION	PHONEERR_NOMEM
PHONEERR_BADDEVICEID	PHONEERR_RESOURCEUNAVAIL
PHONEERR_INVALPOINTER	PHONEERR_OPERATIONFAILED
PHONEERR_NODRIVER	PHONEERR_OPERATIONUNAVAIL

dwDeviceID

Specifies the phone device to be queried.

dwTSPIVersion

Specifies an interface version number that has already been negotiated for this device using **TSPI_phoneNegotiateTSPIVersion**. This function operates according to the interface specification at this version level.

lpExtensionID

Specifies a far pointer to a structure of type **PHONEEXTENSIONID**. If the service provider supports provider-specific extensions, it fills this structure with the extension ID of these extensions. If the service provider does not support extensions, it fills this structure with all zeros. An extension ID of all zeros is not a legal extension ID, since the all-zeros value is used to indicate that the service provider does not support extensions.

This function is typically called by TAPI.DLL in response to an application calling the **phoneNegotiateAPIVersion** function. The result returned by the service provider should be appropriate for use in a subsequent call to **TSPI_phoneNegotiateExtVersion**.

TSPI_phoneGetGain

[New - Windows 95]

```
LONG TSPI_phoneGetGain(HDRVPHONE hdPhone, DWORD dwHookSwitchDev, LPDWORD
lpdwGain)
```

Returns the gain setting of the microphone of the specified phone's hookswitch device.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_RESOURCEUNAVAIL
PHONEERR_INVALPHONESTATE	PHONEERR_OPERATIONFAILED
PHONEERR_INVALHOOKSWITCHDEV	PHONEERR_OPERATIONUNAVAIL
PHONEERR_NOMEM	

hdPhone

Specifies the handle to the phone whose gain is to be retrieved.

dwHookSwitchDev

Identifies the hookswitch device whose gain level is queried. This parameter can only have a single bit set. Values are:

PHONEHOOKSWITCHDEV_HANDSET
: This is the phone's handset.

PHONEHOOKSWITCHDEV_SPEAKER
: This is the phone's speakerphone or adjunct.

PHONEHOOKSWITCHDEV_HEADSET
: This is the phone's headset.

lpdwGain

Specifies a far pointer to a DWORD-sized location into which the service provider writes the current gain setting of the hookswitch microphone component. The *dwGain* gain parameter specifies the volume level of the hookswitch device. This is a number in the range 0x00000000 (which is silence) to 0x0000FFFF (which is maximum volume). The actual granularity and quantization of gain settings in this range are service provider specific.

TSPI_phoneGetHookSwitch

[New - Windows 95]

```
LONG TSPI_phoneGetHookSwitch(HDRVPHONE hdPhone, LPDWORD lpdwHookSwitchDevs)
```

Returns the current hook switch mode of the specified open phone device.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_RESOURCEUNAVAIL
PHONEERR_INVALPHONESTATE	PHONEERR_OPERATIONFAILED
PHONEERR_NOMEM	PHONEERR_OPERATIONUNAVAIL

hdPhone

Specifies the service provider's opaque handle to the phone whose hook switch mode is to be retrieved.

lpdwHookSwitchDevs

Specifies a far pointer to a DWORD-sized location into which the service provider writes the mode of the phone's hookswitch devices. This field uses the PHONEHOOKSWITCHDEV_ constants listed below. If a bit position is FALSE, the corresponding hookswitch device is on hook. If TRUE, the microphone and/or speaker part of the corresponding hookswitch device is offhook. To find out whether microphone and/or speaker are enabled, TAPI.DLL can use **TSPI_phoneGetStatus**. Values for *lpdwHookSwitchDevs* are:

PHONEHOOKSWITCHDEV_HANDSET
This is the phone's handset.

PHONEHOOKSWITCHDEV_SPEAKER
This is the phone's speakerphone or adjunct.

PHONEHOOKSWITCHDEV_HEADSET
This is the phone's headset.

After the hookswitch state of a device changes, and if hookswitch monitoring is enabled, TAPI.DLL is sent a **PHONE_STATE** message.

TSPI_phoneGetIcon

[New - Windows 95]

```
LONG TSPI_phoneGetIcon(DWORD dwDeviceID, LPCSTR lpszDeviceClass, LPHICON
lphIcon)
```

Retrieves a service phone device-specific (or provider-specific) icon for display to the user.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALDEVICECLASS	PHONEERR_OPERATIONFAILED
PHONEERR_NOMEM	PHONEERR_OPERATIONUNAVAIL
PHONEERR_RESOURCEUNAVAIL	

dwDeviceID

Specifies the phone device whose icon is requested.

lpszDeviceClass

Specifies a far pointer to a NULL-terminated ASCII string that identifies a device class name. This device class allows the caller to select a specific sub icon applicable to that device class. This parameter is optional and can be left NULL or be empty, in which case the highest level icon associated with the phone device rather than a specified media stream device would be selected.

lphIcon

Specifies a far pointer to a memory location in which the handle to the icon is returned.

TSPI_phoneGetIcon causes the provider to return a handle (in the DWORD pointed to by *lphIcon*) to an icon resource (obtained from the Windows **LoadIcon** function) associated with the specified phone. The icon handle will be for a resource associated with the provider.

lpszDeviceClass allows the provider to return different icons based on the type of service being referenced by the caller. The permitted strings are the same as for **TSPI_phoneGetID**. For example, if the phone supports the Comm API, passing “COMM” as *lpszDeviceClass* causes the provider to return an icon related specifically to the Comm device functions of the service provider.

The parameters “tapi/phone”, “”, or NULL may be used to request the icon for the phone device. A provider may choose to support many icons (selected by *lpszDeviceClass* and/or phone number), a single icon (such as for the manufacturer, which would be returned for all *phoneGetIcon* requests regardless of the *lpszDeviceClass* selected), or no icons, in which case it sets the DWORD pointed to by *lphIcon* to NULL. TAPI.DLL examines the handle returned by the provider, and if the provider returns NULL, TAPI.DLL substitutes a generic Windows Telephony Icon included as a resource in TAPI.DLL (the generic phone icon).

If the service provider supports no icons, it may leave this function unimplemented, in which case TAPI.DLL will provide a generic “phone” icon for the application.

TSPI_phoneGetID

[New - Windows 95]

```
LONG TSPI_phoneGetID(HDRVPHONE hdPhone, LPVARSTRING lpDeviceID, LPCSTR
lpszDeviceClass)
```

Returns a device ID for the given device class associated with the specified phone device.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_RESOURCEUNAVAIL
PHONEERR_INVALDEVICECLASS	PHONEERR_OPERATIONFAILED
PHONEERR_NOMEM	PHONEERR_OPERATIONUNAVAIL

hdPhone

Specifies the handle to the phone to be queried.

lpDeviceID

Specifies a far pointer to a data structure of type **VARSTRING** where the device ID is returned. The format of the returned information depends on the method used by the device class (API) for naming devices. Prior to calling **TSPI_phoneGetID**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI.DLL for returning information.

lpszDeviceClass

Specifies a far pointer to a NULL-terminated ASCII string that specifies the device class of the device whose ID is requested.

This operation can be used to retrieve a phone device ID given a phone handle. It can also be used to obtain the device ID of the media device (for device classes such as COM, wave, MIDI, phone, line, and mciwave) associated with the opened phone device. This ID can then be used with the appropriate media API (such as mci, midi, and wav) to select the corresponding device.

Some common device class names that are currently used are listed below. The first portion of a name identifies the API used to manage the device class, and the second portion is typically used to identify a specific device type extension or subset of the overall API. Names are not case sensitive:

Device	**Description**
comm	(generic serial-device API; comm port)
comm/datamodem	(reserved for use in a future version of Microsoft Windows)
wave	(low-level waveaudio)
mci/midi	(high-level midi sequencer)
mci/wave	(high-level wave device control)
tapi/line	(TAPI line device)
tapi/phone	(TAPI phone device)
ndis	(network driver interface)

The service provider should fill in all the fields of the **VARSTRING** data structure, except for **dwTotalSize**, which is filled in by TAPI.DLL. The service provider must *not* overwrite the **dwTotalSize** field.

TSPI_phoneGetLamp

[New - Windows 95]

```
LONG TSPI_phoneGetLamp(HDRVPHONE hdPhone, DWORD dwButtonLampID, LPDWORD
lpdwLampMode)
```

Returns the current lamp mode of the specified lamp.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALBUTTONLAMPID	PHONEERR_RESOURCEUNAVAIL
PHONEERR_INVALPHONEHANDLE	PHONEERR_OPERATIONFAILED
PHONEERR_INVALPHONESTATE	PHONEERR_OPERATIONUNAVAIL
PHONEERR_NOMEM	

hdPhone

Specifies the handle to the phone whose lamp mode is to be retrieved.

dwButtonLampID

Specifies the ID of the lamp to be queried.

lpdwLampMode

Specifies a far pointer to a memory location into which the service provider writes the lamp mode status of the given lamp. This parameter can have at most one of the following PHONELAMPMODE_ bits set:

PHONELAMPMODE_BROKENFLUTTER
Broken flutter is the superposition of flash and flutter.

PHONELAMPMODE_FLASH
Flash means slow on and off.

PHONELAMPMODE_FLUTTER
Flutter means fast on and off.

PHONELAMPMODE_OFF
The lamp is off.

PHONELAMPMODE_STEADY
The lamp is continuously lit.

PHONELAMPMODE_WINK
The lamp is winking.

PHONELAMPMODE_UNKNOWN
The lamp mode is currently unknown.

Phone sets that have multiple lamps per button should be modeled using multiple button/lamps pairs. Each additional button/lamp pair should use a DUMMY button.

TSPI_phoneGetRing

[New - Windows 95]

```
LONG TSPI_phoneGetRing(HDRVPHONE hdPhone, LPDWORD lpdwRingMode, LPDWORD
lpdwVolume)
```

Enables an application to query the specified open phone device as to its current ring mode.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_RESOURCEUNAVAIL
PHONEERR_INVALPHONESTATE	PHONEERR_OPERATIONFAILED
PHONEERR_NOMEM	PHONEERR_OPERATIONUNAVAIL

hdPhone

Specifies the handle to the phone whose ring mode is to be queried.

lpdwRingMode

Specifies the ringing pattern with which the phone is ringing. Zero indicates that the phone is not ringing.

lpdwVolume

Specifies the volume level with which the phone is ringing. This is a number in the range 0x00000000 (which is silence) to 0x0000FFFF (which is maximum volume). The actual granularity and quantization of volume settings in this range are service provider specific.

The service provider defines the actual audible ringing patterns corresponding to each of phone's ring modes.

TSPI_phoneGetStatus

[New - Windows 95]

```
LONG TSPI_phoneGetStatus(HDRVPHONE hdPhone, LPPHONESTATUS lpPhoneStatus)
```

This operation queries the specified open phone device for its overall status.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_OPERATIONFAILED
PHONEERR_NOMEM	PHONEERR_OPERATIONUNAVAIL
PHONEERR_RESOURCEUNAVAIL	

hdPhone

Specifies the handle to the phone to be queried.

lpPhoneStatus

Specifies a far pointer to a variably sized data structure of type **PHONESTATUS**, into which the service provider writes information about the phone's status. Prior to calling **TSPI_phoneGetStatus**, the application should set the **dwTotalSize** field of this structure to indicate the amount of memory available to TAPI.DLL for returning information.

The following table indicates which fields of the **PHONESTATUS** data structure are filled in by the service provider and which fields are filled in by TAPI.DLL. The service provider must *not* overwrite the values filled in by TAPI.DLL.

Field Name	TAPI DLL	Service Provider
dwTotalSize;	X	
dwNeededSize;		X
dwUsedSize;		X
dwStatusFlags;		X
dwNumOwners;	X	
dwNumMonitors;	X	
dwRingMode;		X
dwRingVolume;		X
dwHandsetHookSwitchMode;		X
dwHandsetVolume;		X
dwHandsetGain;		X
dwSpeakerHookSwitchMode;		X
dwSpeakerVolume;		X
dwSpeakerGain;		X
dwHeadsetHookSwitchMode;		X
dwHeadsetVolume;		X
dwHeadsetGain;		X
dwDisplaySize;		X
dwDisplayOffset;		X

dwLampModesSize;		X
dwLampModesOffset;		X
dwOwnerNameSize;	X	
dwOwnerNameOffset;	X	
dwDevSpecificSize;		X
dwDevSpecificOffset;		X

TAPI.DLL can use this function to determine the current state of an open phone device. The status information describes information about the phone device's hook switch devices, ringer, volume, display, and lamps of the open phone.

TSPI_phoneGetVolume

[New - Windows 95]

```
LONG TSPI_phoneGetVolume(HDRVPHONE hdPhone, DWORD dwHookSwitchDev, LPDWORD
lpdwVolume)
```

Returns the volume setting of the specified phone's hookswitch device.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_RESOURCEUNAVAIL
PHONEERR_INVALPHONESTATE	PHONEERR_OPERATIONFAILED
PHONEERR_INVALHOOKSWITCHDEV	PHONEERR_OPERATIONUNAVAIL
PHONEERR_NOMEM	

hdPhone

Specifies the handle to the phone containing the hookswitch device whose volume setting is to be retrieved.

dwHookSwitchDev

Identifies a single hook switch device whose volume level is queried. This parameter uses the following PHONEHOOKSWITCHDEV_ constants:

PHONEHOOKSWITCHDEV_HANDSET
: This is the phone's handset.

PHONEHOOKSWITCHDEV_SPEAKER
: This is the phone's speakerphone or adjunct.

PHONEHOOKSWITCHDEV_HEADSET
: This is the phone's headset.

lpdwVolume

Specifies a far pointer to a DWORD-sized location into which the service provider writes the current volume setting of the hookswitch device. This is a number in the range 0x00000000 (which is silence) to 0x0000FFFF (which is maximum volume). The actual granularity and quantization of volume settings in this range are service provider specific.

TSPI_phoneNegotiateExtVersion

[New - Windows 95]

```
LONG TSPI_phoneNegotiateExtVersion(DWORD dwDeviceID, DWORD dwTSPIVersion,
DWORD dwLowVersion, DWORD dwHighVersion, LPDWORD lpdwExtVersion)
```

Returns the highest extension version number the service provider is willing to operate under for this device given the range of possible extension versions.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INCOMPATIBLEAPIVERSION	PHONEERR_RESOURCEUNAVAIL
PHONEERR_INCOMPATIBLEEXTVERSION	PHONEERR_OPERATIONFAILED
PHONEERR_NODRIVER	PHONEERR_OPERATIONUNAVAIL
PHONEERR_NOMEM	

dwDeviceID

Identifies the phone device for which interface version negotiation is to be performed.

dwTSPIVersion

Specifies an interface version number that has already been negotiated for this device using **TSPI_phoneNegotiateTSPIVersion**. This function operates according to the interface specification at this version level.

dwLowVersion

Specifies the lowest extension version number under which TAPI.DLL or its client application is willing to operate. The most-significant WORD is the major version number and the least-significant WORD is the minor version number.

dwHighVersion

Specifies the highest extension version number under which TAPI.DLL or its client application is willing to operate. The most-significant WORD is the major version number and the least-significant WORD is the minor version number.

lpdwExtVersion

Specifies a far pointer to a DWORD. Upon a successful return from this function, the service provider fills this location with the highest extension version number, within the range requested by the caller, under which the service provider is willing to operate. The most-significant WORD is the major version number and the least-significant WORD is the minor version number. If the requested range does not overlap the range supported by the service provider, the function returns PHONEERR_INCOMPATIBLEEXTVERSION.

This function may be called before or after the device has been opened by TAPI.DLL. If the device is currently open and has an extension version selected, the function should return that version number if it is within the requested range. If the selected version number is outside the requested range, the function returns PHONEERR_INCOMPATIBLEEXTVERSION.

TSPI_phoneNegotiateTSPIVersion

[New - Windows 95]

```
LONG TSPI_phoneNegotiateTSPIVersion(DWORD dwDeviceID, DWORD dwLowVersion,
DWORD dwHighVersion, LPDWORD lpdwTSPIVersion)
```

Returns the highest SPI version the service provider is willing to operate under for this device given the range of possible SPI versions.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INCOMPATIBLEAPIVERSION	PHONEERR_RESOURCEUNAVAIL
PHONEERR_NODRIVER	PHONEERR_OPERATIONFAILED
PHONEERR_NOMEM	PHONEERR_OPERATIONUNAVAIL

dwDeviceID

Identifies the phone device for which interface version negotiation is to be performed. Permitted values are strictly within the range of phone devices IDs for this service provider; the value INITIALIZE_NEGOTIATION is never passed to this function.

dwLowVersion

Specifies the lowest TSPI version number under which TAPI.DLL is willing to operate. The most significant WORD is the major version number and the least significant WORD is the minor version number.

dwHighVersion

Specifies the highest TSPI version number under which TAPI.DLL is willing to operate. The most significant WORD is the major version number and the least significant WORD is the minor version number.

lpdwTSPIVersion

Specifies a far pointer to a DWORD. Upon a successful return from this function the service provider fills this location with the highest TSPI version number, within the range requested by the caller, under which the service provider is willing to operate. The most-significant WORD is the major version number and the least-significant WORD is the minor version number. If the requested range does not overlap the range supported by the service provider, the function returns PHONEERR_INCOMPATIBLEAPIVERSION.

The service provider returns PHONEERR_OPERATIONUNAVAIL if the operation is not available. However, if the service provider supports any phone devices, it must also support this function and the function must not return PHONEERR_OPERATIONUNAVAIL.

TAPI.DLL calls this function early in the initialization sequence for each phone device.

Negotiation of an extension version is done through the separate procedure **TSPI_phoneNegotiateExtVersion**.

The corresponding function at the TAPI level is an overloaded function that also retrieves the extension ID, if any, supported by the service provider. At the TSPI level, retrieving the extension ID is accomplished through a separate procedure, namely, **TSPI_phoneGetExtensionID**.

TSPI_phoneOpen

[New - Windows 95]

```
LONG TSPI_phoneOpen(DWORD dwDeviceID, HTAPIPHONE htPhone, LPHDRVPHONE
lphdPhone, DWORD dwTSPIVersion, PHONEEVENT lpfnEventProc)
```

Opens the phone device whose device ID is given, returning the service provider's opaque handle for the device and retaining TAPI.DLL's opaque handle for the device for use in subsequent calls to the **PHONEEVENT** procedure.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_ALLOCATED	PHONEERR_NOMEM
PHONEERR_INCOMPATIBLEAPIVERSION	PHONEERR_RESOURCEUNAVAIL
PHONEERR_NODRIVER	PHONEERR_OPERATIONFAILED
PHONEERR_INUSE	PHONEERR_OPERATIONUNAVAIL
PHONEERR_INIFILECORRUPT	

dwDeviceID

Identifies the phone device to be opened.

htPhone

Specifies TAPI.DLL's opaque handle for the phone device to be used in subsequent calls to the **PHONEEVENT** callback procedure to identify the device.

lphdPhone

A far pointer to an HDRVPHONE where the service provider writes its handle for the phone device to be used by TAPI.DLL in subsequent calls to identify the device.

dwTSPIVersion

The TSPI version negotiated through **TSPI_phoneNegotiateTSPIVersion** under which the service provider is willing to operate.

lpfnEventProc

A far pointer to the **PHONEEVENT** callback procedure supplied by TAPI.DLL that the service provider will call to report subsequent events on the phone.

Opening a phone entitles TAPI.DLL to make further requests on the phone. The phone becomes “active” in the sense that the service provider can report asynchronous events such as hookswitch changes or button presses. The service provider reserves whatever non-sharable resources are required to manage the phone. For example, opening a phone accessed through a comm port and modem should result in opening the comm port, making it no longer available for use by other applications.

If the function is successful, both TAPI.DLL and the service provider become committed to operating under the specified interface version number for this open device. Subsquent operations and events identified using the exchanged opaque phone handles conform to that interface version. This commitment and the validity of the handles remain in effect until TAPI.DLL closes the phone using **TSPI_phoneClose** or until the service provider reports the **PHONE_CLOSE** event. If the function is not successful, no such commitment is made and the handles are not valid.

TSPI_phoneSelectExtVersion

[New - Windows 95]

```
LONG TSPI_phoneSelectExtVersion(HDRVPHONE hdPhone, DWORD dwExtVersion)
```

Selects the indicated extension version for the indicated phone device. Subsequent requests operate according to that extension version.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INCOMPATIBLEEXTVERSION	PHONEERR_OPERATIONFAILED
PHONEERR_NOMEM	PHONEERR_OPERATIONUNAVAIL
PHONEERR_RESOURCEUNAVAIL	

hdPhone

Specifies the handle to the phone for which an extension version is to be selected.

dwExtVersion

Specifies the extension version to be selected. This version number has been negotiated using **TSPI_phoneNegotiateExtVersion**. The most-significant WORD is the major version number and the least-significant WORD is the minor version number. Calling this function with a *dwExtVersion* of zero cancels the current selection.

This function selects the indicated extension version. Although the indicated version number may have been successfully negotiated, a different extension version may have been selected in the interim, in which case this function fails (returning PHONEERR_INCOMPATIBLEEXTVERSION).

Subsequent operations on the phone after an extension version has been selected behave according to that extension version. Subsequent attempts to negotiate the extension version report strictly the selected version or 0 (if the requested range does not include the selected version). Calling this procedure with the special extension version 0 cancels the current selection. The device once again becomes capable of supporting its full range of extension version numbers.

TSPI_phoneSelectExtVersion is typically called in two situations: (1) An application requested to open a phone, the application requested that a particular extension version be used, and no extension version was currently selected, or (2) The last application using a particular extension version closed the phone, and the extension version selection can be cancelled.

TSPI_phoneSetButtonInfo

[New - Windows 95]

```
LONG TSPI_phoneSetButtonInfo(DRV_REQUESTID dwRequestID, HDRVPHONE hdPhone,
DWORD dwButtonLampID, LPPHONEBUTTONINFO const lpButtonInfo)
```

Sets information about the specified button on the specified phone.

- Returns the (positive) *dwRequestID* value if the function will be completed asynchronously, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_RESOURCEUNAVAIL
PHONEERR_INVALBUTTONLAMPID	PHONEERR_OPERATIONFAILED
PHONEERR_NOMEM	PHONEERR_OPERATIONUNAVAIL

dwRequestID

Specifies the identifier of the asynchronous request. The service provider returns this value if the function completes asynchronously.

hdPhone

Specifies the handle to the phone for which button info is to be set.

dwButtonLampID

Specifies a button on the phone device.

lpButtonInfo

Specifies a far pointer to a variably sized structure of type **PHONEBUTTONINFO**. This data structure describes the mode, the function, and provides additional descriptive text to be set for the button.

This function sets the meaning and associated descriptive text of a phone's buttons.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before use to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

TSPI_phoneSetData

[New - Windows 95]

```
LONG TSPI_phoneSetData(DRV_REQUESTID dwRequestID, HDRVPHONE hdPhone, DWORD
dwDataID, LPVOID const lpData, DWORD dwSize)
```

Downloads the information in the specified buffer to the opened phone device at the selected data ID.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_RESOURCEUNAVAIL
PHONEERR_INVALDATAID	PHONEERR_OPERATIONFAILED
PHONEERR_INVALPHONESTATE	PHONEERR_OPERATIONUNAVAIL
PHONEERR_NOMEM	

dwRequestID

Specifies the identifier of the asynchronous request.

hdPhone

Specifies the handle to the phone into which data is to be downloaded.

dwDataID

Specifies where in the phone device the buffer is to be downloaded.

lpData

Specifies a far pointer to the memory location where the data is to be downloaded from.

dwSize

Specifies the size of the buffer in bytes.

The function downloads a maximum of *dwSize* bytes from *lpData* to the phone device. The format of the data, its meaning to the phone device, the meaning of the data ID are service provider specific. The data in the buffer or the selection of a data ID may act as commands to the phone device.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

TSPI_phoneSetDisplay

[New - Windows 95]

```
LONG TSPI_phoneSetDisplay(DRV_REQUESTID dwRequestID, HDRVPHONE hdPhone, DWORD
dwRow, DWORD dwColumn, LPCSTR lpsDisplay, DWORD dwSize)
```

Causes the specified string to be displayed on the specified open phone device.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_RESOURCEUNAVAIL
PHONEERR_INVALPHONESTATE	PHONEERR_OPERATIONFAILED
PHONEERR_INVALPARAM	PHONEERR_OPERATIONUNAVAIL
PHONEERR_NOMEM	

dwRequestID

Specifies the identifier of the asynchronous request.

hdPhone

Specifies the handle to the phone on which the string is to be displayed.

dwRow

Specifies the row on the display where the new text is to be displayed.

dwColumn

Specifies the column position on the display where the new text is to be displayed.

lpsDisplay

Specifies a far pointer to the memory location where the display content is stored. The application should have stored the display information in the format specified as **dwStringFormat** in the phone's PHONECAPS structure.

dwSize

Specifies the size in bytes of the information pointed to by *lpDisplay*.

The specified display information is written to the phone's display, starting at the specified positions. This operation overwrites previously displayed information. If the amount of information exceeds the size of the display, the information will be truncated. The amount of information that can be displayed is at most (**dwNumRows** * **dwNumColumns**) elements in size. The **dwNumRows** and **dwNumColumns** fields are available in the **PHONECAPS** structure returned by **TSPI_phoneGetDevCaps**; they are zero-based.

Although TAPI.DLL checks the validity of pointers passed to this function, this may not be a sufficient test for some service providers due to the asynchronous nature of this function. A service provider that references through these pointers asynchronously (for example, in a different thread after the original thread returns) must recheck pointer validity before using them to ensure that the original thread has not invalidated the pointer(s). If a pointer is no longer valid, the service provider should issue a reply with the LINEERR_INVALPOINTER error value.

TSPI_phoneSetGain

[New - Windows 95]

```
LONG TSPI_phoneSetGain(DRV_REQUESTID dwRequestID, HDRVPHONE hdPhone, DWORD
dwHookSwitchDev, DWORD dwGain)
```

Sets the gain of the microphone of the specified hook switch device to the specified gain level.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_RESOURCEUNAVAIL
PHONEERR_INVALPHONESTATE	PHONEERR_OPERATIONFAILED
PHONEERR_INVALHOOKSWITCHDEV	PHONEERR_OPERATIONUNAVAIL
PHONEERR_NOMEM	

dwRequestID

Specifies the identifier of the asynchronous request.

hdPhone

Specifies the handle to the phone containing the hook switch device whose gain is to be set.

dwHookSwitchDev

Identifies the hook switch device whose microphone's gain is to be set. This parameter can have only a single bit set. Values are:

PHONEHOOKSWITCHDEV_HANDSET
This is the phone's handset.

PHONEHOOKSWITCHDEV_SPEAKER
This is the phone's speakerphone or adjunct.

PHONEHOOKSWITCHDEV_HEADSET
This is the phone's headset.

dwGain

Specifies a far pointer to a DWORD-sized location containing the desired new gain setting of the device. This is a number in the range 0x00000000 (which is silence) to 0x0000FFFF (which is maximum volume). The actual granularity and quantization of gain settings in this range are service-provider-specific. A value for *dwGain* that is out of range is clamped by TAPI.DLL to the nearest in-range value.

TSPI_phoneSetHookSwitch

[New - Windows 95]

```
LONG TSPI_phoneSetHookSwitch(DRV_REQUESTID dwRequestID, HDRVPHONE hdPhone,
DWORD dwHookSwitchDevs, DWORD dwHookSwitchMode)
```

Sets the hook state of the specified open phone's hookswitch devices to the specified mode. Only the hookswitch state of the hookswitch devices listed is affected.

- Returns *dwRequestID* or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_NOMEM
PHONEERR_INVALHOOKSWITCHDEV	PHONEERR_RESOURCEUNAVAIL
PHONEERR_INVALHOOKSWITCHMODE	PHONEERR_OPERATIONFAILED
PHONEERR_INVALPHONESTATE	PHONEERR_OPERATIONUNAVAIL

dwRequestID
Specifies the identifier of the asynchronous request.

hdPhone
Specifies the handle to the phone containing the hookswitch devices whose modes are to be set.

dwHookSwitchDevs
Specifies the device(s) whose hookswitch mode is to be set. This paramter uses the following PHONEHOOKSWITCHDEV_ constants:

PHONEHOOKSWITCHDEV_HANDSET
This is the phone's handset.

PHONEHOOKSWITCHDEV_SPEAKER
This is the phone's speakerphone or adjunct.

PHONEHOOKSWITCHDEV_HEADSET
This is the phone's headset.

dwHookSwitchMode
Specifies the hookswitch mode to set. This parameter can have only one of the following PHONEHOOKSWITCHMODE_ bits set:

PHONEHOOKSWITCHMODE_ONHOOK
The device's microphone and speaker are both onhook.

PHONEHOOKSWITCHMODE_MIC
The device's microphone is active, and the speaker is mute.

PHONEHOOKSWITCHMODE_SPEAKER
The device's speaker is active, and the microphone is mute.

PHONEHOOKSWITCHMODE_MICSPEAKER
The device's microphone and speaker are both active.

The hookswitch mode is changed to the specified setting for all devices specified. If different settings are desired, this function can be invoked multiple times with a different set of parameters. A **PHONE_STATE** message is sent to the application after the hookswitch state has changed.

TSPI_phoneSetLamp

[New - Windows 95]

```
LONG TSPI_phoneSetLamp(DRV_REQUESTID dwRequestID, HDRVPHONE hdPhone, DWORD
dwButtonLampID, DWORD dwLampMode)
```

Causes the specified lamp to be set on the specified open phone device in the specified lamp mode.

- Returns *dwRequestID*, or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_NOMEM
PHONEERR_INVALBUTTONLAMPID	PHONEERR_RESOURCEUNAVAIL
PHONEERR_INVALPHONESTATE	PHONEERR_OPERATIONFAILED
PHONEERR_INVALLAMPMODE	PHONEERR_OPERATIONUNAVAIL

dwRequestID

Specifies the identifier of the asynchronous request.

hdPhone

Specifies the handle to the phone phone whose lamp is to be set.

dwButtonLampID

Identifies the button whose lamp is to be set.

dwLampMode

Specifies how the lamp is to be lit. The *dwLampMode* parameter can only have one of the following PHONELAMPMODE_ bits set:

PHONELAMPMODE_BROKENFLUTTER
Broken flutter is the superposition of flash and flutter.

PHONELAMPMODE_FLASH
Flash means slow on and off.

PHONELAMPMODE_FLUTTER
Flutter means fast on and off.

PHONELAMPMODE_OFF
The lamp is off.

PHONELAMPMODE_STEADY
The lamp is continuously lit.

PHONELAMPMODE_WINK
The lamp is winking.

TSPI_phoneSetRing

[New - Windows 95]

```
LONG TSPI_phoneSetRing(DRV_REQUESTID dwRequestID, HDRVPHONE hdPhone, DWORD
dwRingMode, DWORD dwVolume)
```

Rings the specified open phone device using the specified ring mode and volume.

- Returns *dwRequestID* or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_RESOURCEUNAVAIL
PHONEERR_INVALPHONESTATE	PHONEERR_OPERATIONFAILED
PHONEERR_INVALRINGMODE	PHONEERR_OPERATIONUNAVAIL
PHONEERR_NOMEM	

dwRequestID

Specifies the identifier of the asynchronous request.

hdPhone

Specifies the handle to the phone to be rung.

dwRingMode

Specifies the ringing pattern with which to ring the phone. This parameter must be within the range of 0 to the value of the **dwNumRingModes** field in the **PHONECAPS** structure. If **dwNumRingModes** is 0, the ring mode of the phone cannot be controlled; if dwNumRingModes is 1, a value of 0 for *dwRingMode* indicates that the phone should not be rung (silence), and other values from 1 to **dwNumRingModes** are valid ring modes for the phone device.

dwVolume

Specifies the volume level with which the phone is to be rung. This is a number in the range 0x00000000 (which is silence) to 0x0000FFFF (which is maximum volume). The actual granularity and quantization of volume settings in this range are service provider specific. A value for *dwVolume* that is out of range is clamped by TAPI.DLL to the nearest value in range.

The service provider defines the actual audible ringing patterns corresponding to each of the phone's ring modes.

TSPI_phoneSetStatusMessages

[New - Windows 95]

```
LONG TSPI_phoneSetStatusMessages(HDRVPHONE hdPhone, DWORD dwPhoneStates,
DWORD dwButtonModes, DWORD dwButtonStates)
```

This operation causes the service provider to filter status messages that are not currently of interest to any application.

- Returns zero if the function is successful, or a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_NOMEM
PHONEERR_INVALPHONESTATE	PHONEERR_RESOURCEUNAVAIL
PHONEERR_INVALBUTTONMODE	PHONEERR_OPERATIONFAILED
PHONEERR_INVALBUTTONSTATE	PHONEERR_OPERATIONUNAVAIL

hdPhone

Specifies the opaque handle to the phone whose state-change monitoring filter is to be set.

dwPhoneStates

These flags specify the set of phone status changes and events for which TAPI.DLL wishes to receive notification messages. This parameter can have zero, one, or more than one of the following PHONESTATE_ bits set:

PHONESTATE_OTHER
A phone status information item other than those listed below has changed.

PHONESTATE_CONNECTED
The connection between the phone device and the service provider was just made. This happens when the API is first invoked, or when the wire connecting the phone to the computer is plugged in with the API active.

PHONESTATE_DISCONNECTED
The connection between the phone device and the service provider was just broken. This happens when the wire connecting the phone set to the computer is unplugged while the service provider is active.

PHONESTATE_DISPLAY
The display of the phone has changed.

PHONESTATE_LAMP
A lamp of the phone has changed.

PHONESTATE_RINGMODE
The ring mode of the phone has changed.

PHONESTATE_RINGVOLUME
The ring volume of the phone has changed.

PHONESTATE_HANDSETHOOKSWITCH
The handset hookswitch state has changed.

PHONESTATE_HANDSETVOLUME
The handset's speaker volume setting has changed.

PHONESTATE_HANDSETGAIN
The handset's microphone gain setting has changed.

PHONESTATE_SPEAKERHOOKSWITCH
The speakerphone's hookswitch state has changed.

PHONESTATE_SPEAKERVOLUME
The speakerphone's speaker volume setting has changed.

PHONESTATE_SPEAKERGAIN
The speakerphone's microphone gain setting state has changed.

PHONESTATE_HEADSETHOOKSWITCH
The headset's hookswitch state has changed.

PHONESTATE_HEADSETVOLUME
The headset's speaker volume setting has changed.

PHONESTATE_HEADSETGAIN
The headset's microphone gain setting has changed.

PHONESTATE_SUSPEND
TAPI.DLL's use of the phone is temporarily suspended. This may occur when the switch requires exclusive use of the phone device.

PHONESTATE_RESUME
TAPI.DLL's use of the phone device is resumed after having been suspended for some time.

PHONESTATE_DEVSPECIFIC
The phone's device-specific information has changed.

PHONESTATE_REINIT
Items have changed in the configuration of phone devices. To become aware of these changes, (as with the appearance of new phone devices) the application should reinitialize its use of TAPI. If specified, the *htPhone* parameter is left NULL for this state change as it applies to any of the phones in the system.

The **PHONESTATE_REINIT** message is never masked. REINIT messages should always be sent when appropriate, regardless of the setting of this flag. The setting of the REINIT flag is simply ignored and generates no error.

dwButtonModes

These flags specify the set of phone button modes for which TAPI.DLL wishes to receive notification messages. Note that if *dwButtonModes* is 0, *dwButtonStates* is ignored. This parameter can have zero, one, or more than one of the following PHONEBUTTONMODE_ bits set. If *dwButtonModes* has at least one of these flags set, *dwButtonStates* must also have at least one bit set.

PHONEBUTTONMODE_CALL
The button is assigned to a call appearance.

PHONEBUTTONMODE_FEATURE
The button is assigned to requesting features from the switch, such as hold, conference, or transfer.

PHONEBUTTONMODE_KEYPAD
The button is one of the twelve keypad buttons, '0' through '9', '*', and '#'.

PHONEBUTTONMODE_LOCAL
The button is a local function button, such as mute or volume control.

PHONEBUTTONMODE_DISPLAY
The button is a soft button associated with the phone's display. A phone set can have zero or more display buttons.

dwButtonStates

This parameter specifies the set of phone button state changes for which TAPI.DLL wishes to receive notification messages. Values are:

PHONEBUTTONSTATE_UP
The button is in the up state.

PHONEBUTTONSTATE_DOWN
The button is in the down state (pressed down).

TAPI defines a number of messages that notify applications about events occurring on phones. The sets of all change messages in which all applications are interested may be much smaller than the set of possible messages. This procedure allows TAPI.DLL to tell the service provider the reduced set of messages that should be delivered. The service provider should deliver all of the messages it supports, within the specified set. It is permitted to deliver more (they will be filtered out by TAPI.DLL), but is discouraged from doing so for performance reasons. If TAPI.DLL requests delivery of a particular message type that is not produced by the provider, the provider should nevertheless accept the request but simply not produce the message. All phone status messages except **PHONESTATE_REINIT** are disabled by default.

TSPI_phoneSetVolume

[New - Windows 95]

```
LONG TSPI_phoneSetVolume(DRV_REQUESTID dwRequestID, HDRVPHONE hdPhone, DWORD
dwHookSwitchDev, DWORD dwVolume)
```

Either sets the volume of the speaker component of the specified hookswitch device to the specified level.

- Returns *dwRequestID* or a negative error number if an error has occurred. The *lResult* actual parameter of the corresponding **ASYNC_COMPLETION** is zero if the function is successful or it is a negative error number if an error has occurred. Possible return values are as follows:

PHONEERR_INVALPHONEHANDLE	PHONEERR_RESOURCEUNAVAIL
PHONEERR_INVALPHONESTATE	PHONEERR_OPERATIONFAILED
PHONEERR_INVALHOOKSWITCHDEV	PHONEERR_OPERATIONUNAVAIL
PHONEERR_NOMEM	

dwRequestID

Specifies the identifier of the asynchronous request.

hdPhone

Specifies the handle to the phone containing the speaker whose volume is to be set.

dwHookSwitchDev

Identifies the hook switch device whose speaker's volume is to be set. This parameter uses the following PHONEHOOKSWITCHDEV_ constants:

PHONEHOOKSWITCHDEV_HANDSET
This is the phone's handset.

PHONEHOOKSWITCHDEV_SPEAKER
This is the phone's speakerphone or adjunct.

PHONEHOOKSWITCHDEV_HEADSET
This is the phone's headset.

dwVolume

A DWORD specifying the new volume level of the hookswitch device. This is a number in the range 0x00000000 (which is silence) to 0x0000FFFF (which is maximum volume). The actual granularity and quantization of volume settings in this range are service provider specific. A value for *dwVolume* that is out of range is clamped by TAPI.DLL to the nearest value in range.

Phone Device Messages

Phone Device Messages

About Phone Device Messages

The PHONE_REPLY message is not used at the TSPI level. Completion of an asynchronous request is reported using the **ASYNC_COMPLETION** callback.

Reference

PHONE_BUTTON

[New - Windows 95]

```
htPhone = (HTAPIPHONE) hPhoneDevice;
dwMsg = (DWORD) PHONE_BUTTON;
dwParam1 = (DWORD) idButtonLamp;
dwParam2 = (DWORD) PhoneButtonMode;
dwParam3 = (DWORD) PhoneButtonState;
```

Sent to the **PHONEEVENT** callback to notify TAPI.DLL that it has detected a button press on a phone device that TAPI.DLL has open and for which button press monitoring is enabled.

htPhone

Specifies TAPI.DLL's opaque object handle to the phone device.

dwMsg

The value PHONE_BUTTON

dwParam1

Specifies the button/lamp ID of the button that was pressed. Note that the button IDs 0 through 11 are always the KEYPAD buttons. '0' is button ID 0, '1' is button ID 1, and so on, through button ID 9. '*' is button ID 10, and '#' is button ID 11. Additional information about a button ID is available through **TSPI_phoneGetDevCaps**, and **TSPI_phoneGetButtonInfo**.

dwParam2

Specifies the button mode of the button. This parameter uses the following PHONEBUTTONMODE_ constants:

PHONEBUTTONMODE_CALL
The button is assigned to a call appearance.

PHONEBUTTONMODE_FEATURE
The button is assigned to requesting features from the switch, such as hold, conference, or transfer.

PHONEBUTTONMODE_KEYPAD
The button is one of the twelve keypad buttons, '0' through '9', '*', and '#'.

PHONEBUTTONMODE_LOCAL
The button is a local function button, such as mute or volume control.

PHONEBUTTONMODE_DISPLAY
The button is a "soft" button associated with the phone's display. A phone set can have zero or more display buttons.

dwParam3

Specifies whether this is a button down event or a button up event. This parameter uses the following PHONEBUTTONSTATE_ constants:

PHONEBUTTONSTATE_UP
The button is in the "up" state.

PHONEBUTTONSTATE_DOWN
The button is in the "down" state (pressed down).

The set of phone button events that is reported is selected by the function **TSPI_phoneSetStatusMessages**. The service provider must send **PHONE_BUTTON** messages for at least the set of button press events selected through that procedure. The service provider may send more than this set but it should try to limit its messages to this set for performance reasons.

A **PHONE_BUTTON** message is sent whenever a button changes state. The service provider must guarantee that each button DOWN event is strictly paired with a button UP event, in that order. Note that a service provider that can detect only one of the button press or release events should expand the event it does detect into a button DOWN message followed by a button UP message.

PHONE_CLOSE

[New - Windows 95]

```
htPhone = (HTAPIPHONE) hPhoneDevice;
dwMsg = (DWORD) PHONE_CLOSE;
dwParam1 = (DWORD) 0;
dwParam2 = (DWORD) 0;
dwParam3 = (DWORD) 0;
```

Sent to the **PHONEEVENT** callback when an open phone device has been forcibly closed. The phone device handle is no longer valid once this message has been sent. The service provider guarantees that all asynchronous requests on the phone have been reported complete by calling **ASYNC_COMPLETION** for each outstanding request before this message is sent. TAPI DLL must not request any future operations using this phone handle.

htPhone

Specifies TAPI.DLL's opaque object handle to the phone device.

dwMsg

The value PHONE_CLOSE

dwParam1

Unused.

dwParam2

Unused.

dwParam3

Unused.

This message is only sent to TAPI DLL after an open phone has been forcibly closed. This may be done, for example, when taking the phone out of service or reconfiguring the service provider. Whether or not the phone can be reopened immediately after a forced close is up to the service provider.

An open phone device may be forcibly closed after the user has modified the configuration of that phone or its driver. If the user wants the configuration changes to be effective immediately (instead of after the next system restart), and they affect the application's current view of the device (such as a change in device capabilities), a service provider may forcibly close the phone device.

PHONE_DEVSPECIFIC

[New - Windows 95]

```
htPhone = (HTAPIPHONE) hPhoneDevice;
dwMsg = (DWORD) PHONE_DEVSPECIFIC;
dwParam1 = (DWORD) DeviceData1;
dwParam2 = (DWORD) DeviceData2;
dwParam3 = (DWORD) DeviceData3;
```

Sent to the **PHONEEVENT** callback to notify TAPI.DLL about device-specific events occurring at the phone. The meaning of the message and the interpretation of the parameters is implementation-defined.

htPhone

Specifies TAPI.DLL's opaque object handle to the phone device.

dwMsg

The value PHONE_DEVSPECIFIC

dwParam1

Device specific.

dwParam2

Device specific.

dwParam3

Device specific.

TAPI.DLL sends the **PHONE_DEVSPECIFIC** message to applications in response to receiving this message from a service provider. The *htPhone* is translated to the appropriate *hPhone* as the *hDevice* parameter at the TAPI level. The *dwParam1, dwParam2,* and *dwParam3* paramters are passed through unmodified.

PHONE_STATE

[New - Windows 95]

```
htPhone = (HTAPIPHONE) hPhoneDevice;
dwMsg = (DWORD) PHONE_STATE;
dwParam1 = (DWORD) PhoneState;
dwParam2 = (DWORD) PhoneStateData;
dwParam3 = (DWORD) 0;
```

Sent to the **PHONEEVENT** callback to TAPI.DLL whenever a phone device's status changes.

htPhone

Specifies TAPI.DLL's opaque object handle to the phone device.

dwMsg

The value PHONE_STATE

dwParam1

Specifies the phone state that has changed. This parameter uses the following PHONESTATE_ constants:

PHONESTATE_OTHER
A phone status information item other than those listed below has changed.

PHONESTATE_CONNECTED
The connection between the phone device and TAPI.DLL was just made. This happens when TAPI.DLL is first invoked, or when the wire connecting the phone to the computer is plugged in with TAPI.DLL active.

PHONESTATE_DISCONNECTED
The connection between the phone device and TAPI.DLL was just broken. This happens when the wire connecting the phone set to the computer is unplugged while TAPI.DLL is active.

PHONESTATE_DISPLAY
The display of the phone has changed.

PHONESTATE_LAMP
A lamp of the phone has changed.

PHONESTATE_RINGMODE
The ring mode of the phone has changed.

PHONESTATE_RINGVOLUME
The ring volume of the phone has changed.

PHONESTATE_HANDSETHOOKSWITCH
The handset hookswitch state has changed.

PHONESTATE_HANDSETVOLUME
The handset's speaker volume setting has changed.

PHONESTATE_HANDSETGAIN
The handset's mic gain setting has changed.

PHONESTATE_SPEAKERHOOKSWITCH
The speakerphone's hookswitch state has changed.

PHONESTATE_SPEAKERVOLUME
The speakerphone's speaker volume setting has changed.

PHONESTATE_SPEAKERGAIN
The speakerphone's mic gain setting state has changed.

PHONESTATE_HEADSETHOOKSWITCH
The headset's hookswitch state has changed.

PHONESTATE_HEADSETVOLUME
The headset's speaker volume setting has changed.

PHONESTATE_HEADSETGAIN
The headset's mic gain setting has changed.

PHONESTATE_SUSPEND
The application's use of the phone is temporarily suspended.

PHONESTATE_RESUME

The application's use of the phone device is resumed after having been suspended for some time.

PHONESTATE_DEVSPECIFIC
The phone's device-specific information has changed.

PHONESTATE_REINIT
Items have changed in the configuration of phone devices. To become aware of these changes (as with the appearance of new phone devices), the application should reinitialize its use of TAPI. If specified, the *htPhone* parameter is left NULL for this state change as it applies to any of the phones in the system.

dwParam2

Specifies phone-state-dependent information detailing the status change. This field is not used if multiple flags are set in *dwParam1*, since multiple status items have changed. TAPI.DLL should invoke **TSPI_phoneGetStatus** to obtain a complete set of information. If *dwParam1* is PHONESTATE_LAMP, *dwParam2* contains the button/lamp ID of the lamp that has changed. If *dwParam1* is PHONESTATE_RINGMODE, *dwParam2* contains the new ring mode. If *dwParam1* is PHONESTATE_HANDSETHOOKSWITCH, PHONESTATE_SPEAKERHOOKSWITCH or PHONESTATE_HEADSETHOOKSWITCH, *dwParam2* contains the new hookswitch mode of that hookswitch device, which can be set to any of the following PHONEHOOKSWITCHMODE_ constants:

PHONEHOOKSWITCHMODE_ONHOOK
The device's mic and speaker are both onhook.

PHONEHOOKSWITCHMODE_MIC
The device's mic is active, the speaker is mute.

PHONEHOOKSWITCHMODE_SPEAKER
The device's speaker is active, the mic is mute.

PHONEHOOKSWITCHMODE_MICSPEAKER
The device's mic and speaker are both active.

dwParam3

Unused.

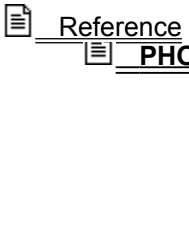
The sending of this message is controlled by **TSPI_phoneSetStatusMessages**. By default, this message is disabled for all state changes.

The service provider never reports changed values for PHONESTATE_OWNER or PHONESTATE_MONITORS, since the notion of privilege does not appear at the TSPI level.

Phone Device Structures

Reference

Phone Device Structures



Reference

PHONEBUTTONINFO

[New - Windows 95]

```
typedef struct phonebuttoninfo_tag {
  DWORD  dwTotalSize;
  DWORD  dwNeededSize;
  DWORD  dwUsedSize;

  DWORD  dwButtonMode;
  DWORD  dwButtonFunction;

  DWORD  dwButtonTextSize;
  DWORD  dwButtonTextOffset;

  DWORD  dwDevSpecificSize;
  DWORD  dwDevSpecificOffset;
} PHONEBUTTONINFO, FAR *LPPHONEBUTTONINFO;
```

The **PHONEBUTTONINFO** structure contains information about a button on a phone device.

dwTotalSize
The total size in bytes allocated to this data structure.

dwNeededSize
The size in bytes for this data structure that is needed to hold all the returned information.

dwUsedSize
The size in bytes of the portion of this data structure that contains useful information.

dwButtonMode
Defines the mode or general usage class of the button. This field uses the following PHONEBUTTONMODE_ constants:

PHONEBUTTONMODE_DUMMY
This value is used to describe a button/lamp position that has no corresponding button, but has only a lamp.

PHONEBUTTONMODE_CALL
The button is assigned to a call appearance.

PHONEBUTTONMODE_FEATURE
The button is assigned to requesting features from the switch, such as hold, conference, and transfer.

PHONEBUTTONMODE_KEYPAD
The button is one of the twelve keypad buttons, ‘0’ through ‘9’, ‘*’, and ‘#’.

PHONEBUTTONMODE_LOCAL
The button is a local function button, such as mute or volume control.

PHONEBUTTONMODE_DISPLAY
The button is a “soft” button associated with the phone’s display. A phone set can have zero or more display buttons.

dwButtonFunction
Specifies the function assigned to the button. This field uses the PHONEBUTTONFUNCTION_ constants (not listed here).

dwButtonTextSize
dwButtonTextOffset
The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized field containing descriptive text for this button. The format of this information is as specified in the **dwStringFormat** field of the phone’s device capabilities.

dwDevSpecificSize
dwDevSpecificOffset
The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device-specific field.

Device-specific extensions should use the DevSpecific (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

PHONECAPS

[New - Windows 95]

```
typedef struct phonecaps_tag {
  DWORD  dwTotalSize;
  DWORD  dwNeededSize;
  DWORD  dwUsedSize;

  DWORD  dwProviderInfoSize;
  DWORD  dwProviderInfoOffset;

  DWORD  dwPhoneInfoSize;
  DWORD  dwPhoneInfoOffset;

  DWORD  dwPermanentPhoneID;
  DWORD  dwPhoneNameSize;
  DWORD  dwPhoneNameOffset;
  DWORD  dwStringFormat;

  DWORD  dwPhoneStates;
  DWORD  dwHookSwitchDevs;
  DWORD  dwHandsetHookSwitchModes;
  DWORD  dwSpeakerHookSwitchModes;
  DWORD  dwHeadsetHookSwitchModes;

  DWORD  dwVolumeFlags;
  DWORD  dwGainFlags;
  DWORD  dwDisplayNumRows;
  DWORD  dwDisplayNumColumns;
  DWORD  dwNumRingModes;
  DWORD  dwNumButtonLamps;

  DWORD  dwButtonModesSize;
  DWORD  dwButtonModesOffset;

  DWORD  dwButtonFunctionsSize;
  DWORD  dwButtonFunctionsOffset;
  DWORD  dwLampModesSize;
  DWORD  dwLampModesOffset;

  DWORD  dwNumSetData;
  DWORD  dwSetDataSize;
  DWORD  dwSetDataOffset;

  DWORD  dwNumGetData
  DWORD  dwGetDataSize;
  DWORD  dwGetDataOffset;

  DWORD  dwDevSpecificSize;
  DWORD  dwDevSpecificOffset;


} PHONECAPS, FAR *LPPHONECAPS;
```

The **PHONECAPS** structure describes the capabilities of a phone device.

dwTotalSize
The total size in bytes allocated to this data structure.

dwNeededSize
The size in bytes for this data structure that is needed to hold all the returned information.

dwUsedSize

The size in bytes of this data structure that contains useful information.

dwProviderInfoSize
dwProviderInfoOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized field containing service-provider-specific information. The **dwProviderInfoSize/Offset** field is intended to provide information about the provider hardware and/or software, such as the vendor name and version numbers of hardware and software. This information can be useful when a user needs to call customer service with problems regarding the provider.

dwPhoneInfoSize
dwPhoneInfoOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device field containing phone-specific information. The **dwPhoneInfoSize/Offset** field is intended to provide information about the attached phone device, such as the phone-device manufacturer, the model name, the software version, and so on. This information can be useful when a user needs to call customer service with problems regarding the phone.

dwPermanentPhoneID

Specifies the permanent DWORD identifier by which the phone device is known in the system's configuration.

dwPhoneNameSize
dwPhoneNameOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device field containing user configurable name for this phone device. This name can be configured by the user when configuring the phone device's service provider and is provided for the user's convenience.

dwStringFormat

Specifies the string format to be used with this phone device. This field uses the following STRINGFORMAT_ constants:

STRINGFORMAT_ASCII
This is ASCII string format using one byte per character.

STRINGFORMAT_DBCS
This is DBCS string format using two bytes per character.

STRINGFORMAT_UNICODE
This is unicode string format using two bytes per character.

dwPhoneStates

Specifies the state changes for this phone device for which the application can be notified via a **PHONE_STATE** callback message. This field uses the following PHONESTATE_ constants:

PHONESTATE_OTHER
A phone status information item other than those listed below has changed.

PHONESTATE_CONNECTED
The connection between the phone device and TAPI.DLL was just made. This happens when TAPI.DLL is first invoked, or when the wire connecting the phone to the computer is plugged in when TAPI.DLL is active.

PHONESTATE_DISCONNECTED
The connection between the phone device and TAPI.DLL was just broken. This happens when the wire connecting the phone set to the computer is unplugged while TAPI.DLL is active.

PHONESTATE_OWNER
The number of owners for the phone device has changed.

PHONESTATE_MONITORS
The number of monitors for the phone device has changed.

PHONESTATE_DISPLAY
The display of the phone has changed.

PHONESTATE_LAMP
A lamp of the phone has changed.

PHONESTATE_RINGMODE

The ring mode of the phone has changed.

PHONESTATE_RINGVOLUME
The ring volume of the phone has changed.

PHONESTATE_HANDSETHOOKSWITCH
The handset hookswitch state has changed.

PHONESTATE_HANDSETVOLUME
The handset's speaker volume setting has changed.

PHONESTATE_HANDSETGAIN
The handset's microphone gain setting has changed.

PHONESTATE_SPEAKERHOOKSWITCH
The speakerphone's hookswitch state has changed.

PHONESTATE_SPEAKERVOLUME
The speakerphone's speaker volume setting has changed.

PHONESTATE_SPEAKERGAIN
The speakerphone's microphone gain setting state has changed.

PHONESTATE_HEADSETHOOKSWITCH
The headset's hookswitch state has changed.

PHONESTATE_HEADSETVOLUME
The headset's speaker volume setting has changed.

PHONESTATE_HEADSETGAIN
The headset's microphone gain setting has changed.

PHONESTATE_SUSPEND
The application's use of the phone is temporarily suspended.

PHONESTATE_RESUME
The application's use of the phone device is resumed after having been suspended for some time.

PHONESTATE_DEVSPECIFIC
The phone's device-specific information has changed.

PHONESTATE_REINIT
Items have changed in the configuration of phone devices. To become aware of these changes (such as for the appearance of new phone devices) the application should reinitialize its use of TAPI. If specified, the *htPhone* parameter is left NULL for this state change as it applies to any of the phones in the system.

dwHookSwitchDevs

This field specifies the phone's hookswitch devices, and uses the following PHONEHOOKSWITCHDEV_ constants:

PHONEHOOKSWITCHDEV_HANDSET
This is the ubiquitous, hand-held ear and mouth piece.

PHONEHOOKSWITCHDEV_SPEAKER
A built-in loudspeaker and microphone. This could also be an externally connected adjunct to the telephone set.

PHONEHOOKSWITCHDEV_HEADSET
This is a headset connected to the phone set.

dwHandsetHookSwitchModes
dwSpeakerHookSwitchModes
dwHeadsetHookSwitchModes

These fields specifies the phone's hookswitch mode capabilities of the handset, speaker, or headset respectively. They are only meaningful if the hookswitch device is listed in **dwHookSwitchDevs**. The following PHONEHOOKSWITCHMODE_ constants are used:

PHONEHOOKSWITCHMODE_ONHOOK
The device's microphone and speaker are both onhook.

PHONEHOOKSWITCHMODE_MIC
The device's microphone is active and the speaker is mute.

PHONEHOOKSWITCHMODE_SPEAKER
The device's speaker is active and the microphone is mute.

PHONEHOOKSWITCHMODE_MICSPEAKER

The device's microphone and speaker are both active.

PHONEHOOKSWITCHMODE_UNKNOWN
The device's hookswitch state is unknown or cannot be detected.

dwVolumeFlags
This field specifies the volume setting capabilities of the phone device's speaker components. If the bit in position PHONEHOOKSWITCHDEV_ is TRUE, the volume of the corresponding hookswitch device's speaker component can be adjusted via **TSPI_phoneSetVolume**; otherwise FALSE.

dwGainFlags
This field specifies the gain setting capabilities of the phone device's microphone components. If the bit position PHONEHOOKSWITCHDEV_ is TRUE, the volume of the corresponding hookswitch device's microphone component can be adjusted using **TSPI_phoneSetGain**; otherwise FALSE.

dwDisplayNumRows
This field specifies the display capabilities of the phone device by describing the number of rows in the phone display. **dwDisplayNumRows** and **dwDisplayNumColumns** are both zero for a phone device without a display.

dwDisplayNumColumns
This field specifies the display capabilities of the phone device by describing the number of columns in the phone display. **dwDisplayNumRows** and **dwDisplayNumColumns** are both zero for a phone device without a display.

dwNumRingModes
This field specifies the ring capabilities of the phone device. The phone is able to ring with **dwNumRingModes** different ring patterns, identified as 1, 2, ... **dwNumRingModes** minus one. If the value of this field is zero, applications have no control over the ring mode of the phone. If the value of this field is greater than zero, it indicates the number of ring modes in addition to silence that are supported by the service provider. A value of 0 in the *lpdwRingMode* parameter of **TSPI_phoneGetRing** or the *dwRingMode* parameter of **TSPI_phoneSetRing** indicates silence (the phone is not ringing or should not be rung), and *dwRingMode* values of 1 to **dwNumRingModes** are valid ring modes for the phone device.

dwNumButtonLamps
This field specifies the number of button/lamps on the phone device that are detectable in TAPI. Button/lamps are identified by their ID. Valid button/lamp IDs range from zero to **dwNumButtonLamps** minus one. The keypad buttons '0', through '9', '*', and '#' are assigned the IDs 0 through 12.

dwButtonModesSize
dwButtonModesOffset
The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field containing the button modes of the phone's buttons, of type PHONEBUTTONMODE_. The array is indexed by button/lamp ID. Values are:

PHONEBUTTONMODE_DUMMY
This value is used to describe a button/lamp position that has no corresponding button, but has only a lamp. If the phone provides any non-DUMMY buttons, the **PHONE_BUTTON** message will be sent to the application if a button is pressed at the phone device.

PHONEBUTTONMODE_CALL
The button is assigned to a call appearance.

PHONEBUTTONMODE_FEATURE
The button is assigned to requesting features from the switch, such as hold, conference, and transfer.

PHONEBUTTONMODE_KEYPAD
The button is one of the twelve keypad buttons, '0' through '9', '*', and '#'.

PHONEBUTTONMODE_LOCAL
The button is a local function button, such as mute or volume control.

PHONEBUTTONMODE_DISPLAY
The button is a "soft" button associated with the phone's display. A phone set can have zero or more display buttons.

dwButtonFunctionsSize
dwButtonFunctionsOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field containing the button modes of the phone's buttons, of type PHONEBUTTONFUNCTION_. The array is indexed by button/lamp ID.

dwLampModesSize
dwLampModesOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field containing the lamp modes of the phone's lamps, of type PHONELAMPMODE_. The array is indexed by button/lamp ID. Values are:

PHONELAMPMODE_BROKENFLUTTER
Broken flutter is the superposition of flash and flutter.

PHONELAMPMODE_FLASH
Flash means slow on and off.

PHONELAMPMODE_FLUTTER
Flutter means fast on and off.

PHONELAMPMODE_OFF
The lamp is off.

PHONELAMPMODE_STEADY
The lamp is continuously lit.

PHONELAMPMODE_WINK
The lamp is winking.

PHONELAMPMODE_DUMMY
This value is used to describe a button/lamp position that has no corresponding lamp.

dwNumSetData

The number of different download areas in the phone device. The different areas are referred to using the data IDs 0, 1, ..., **dwNumSetData** minus one. If this field is zero, the phone does not support the download capability.

dwSetDataSize
dwSetDataOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field containing the sizes (in bytes) of the phone's download data areas. This is an array with DWORD-sized elements, indexed by data ID.

dwNumGetData

The number of different upload areas in the phone device. The different areas are referred to using the data IDs 0, 1, ..., **dwNumGetData** minus one. If this field is zero, the phone does not support the upload capability.

dwGetDataSize
dwGetDataOffset

The size in bytes and the offset from the beginning of this data structure in bytes of the variably sized field containing the sizes (in bytes) of the phone's upload data areas. This is an array with DWORD-sized elements, indexed by data ID.

dwDevSpecificSize
dwDevSpecificOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device-specific field.

Device-specific extensions should use the DevSpecific (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

PHONEEXTENSIONID

[New - Windows 95]

```
typedef struct phoneextensionid_tag {
  DWORD  dwExtensionID0;
  DWORD  dwExtensionID1;
  DWORD  dwExtensionID2;
  DWORD  dwExtensionID3;
} PHONEEXTENSIONID, FAR *LPPHONEEXTENSIONID;
```

The **PHONEEXTENSIONID** structure describes an extension ID. Extension IDs are used to identify service-provider-specific extensions for phone device classes.

dwExtensionID0
dwExtensionID1
dwExtensionID2
dwExtensionID3

These four DWORD-sized fields together specify a universally unique extension ID that identifies a phone device class extension.

Not extensible.

Extension IDs are generated using an SDK-provided generation utility.

PHONESTATUS

[New - Windows 95]

```
typedef struct phonestatus_tag {
  DWORD  dwTotalSize;
  DWORD  dwNeededSize;
  DWORD  dwUsedSize;

  DWORD  dwStatusFlags;
  DWORD  dwNumOwners;
  DWORD  dwNumMonitors;
  DWORD  dwRingMode;
  DWORD  dwRingVolume;

  DWORD  dwHandsetHookSwitchMode;
  DWORD  dwHandsetVolume;
  DWORD  dwHandsetGain;

  DWORD  dwSpeakerHookSwitchMode;
  DWORD  dwSpeakerVolume;
  DWORD  dwSpeakerGain;

  DWORD  dwHeadsetHookSwitchMode;
  DWORD  dwHeadsetVolume;
  DWORD  dwHeadsetGain;

  DWORD  dwDisplaySize;
  DWORD  dwDisplayOffset;

  DWORD  dwLampModesSize;
  DWORD  dwLampModesOffset;

  DWORD  dwOwnerNameSize;
  DWORD  dwOwnerNameOffset;

  DWORD  dwDevSpecificSize;
  DWORD  dwDevSpecificOffset;
} PHONESTATUS, FAR *LPPHONESTATUS;
```

The **PHONESTATUS** structure describes the current status of a phone device.

dwTotalSize
The total size in bytes allocated to this data structure.

dwNeededSize
The size in bytes for this data structure that is needed to hold all the returned information.

dwUsedSize
The size in bytes of the portion of this data structure that contains useful information.

dwStatusFlags
This field provides a collection of status flags for this phone device. The PHONESTATUSFLAGS_ constants used by this field are:

PHONESTATUSFLAGS_CONNECTED
Specifies whether the phone is currently connected to TAPI. TRUE if connected, FALSE otherwise.

PHONESTATUSFLAGS_SUSPENDED
Specifies whether TAPI's manipulation of the phone device is suspended. TRUE if suspended; otherwise FALSE. An application's use of a phone device may be temporarily suspended when the switch wants to manipulate the phone in a way that cannot tolerate interference from the application.

dwRingMode

Specifies the current ring mode of a phone device.

dwRingVolume

Specifies the current ring volume of a phone device. This is a value between 0x00000000 (silence) and 0x0000FFFF (maximum volume).

dwHandsetHookSwitchMode

Specifies the current hook switch mode of the phone's handset. This field uses the following PHONEHOOKSWITCHMODE_ constants:

PHONEHOOKSWITCHMODE_ONHOOK
The device's microphone and speaker are both onhook.

PHONEHOOKSWITCHMODE_MIC
The devicer's microphone is active, the speaker is mute.

PHONEHOOKSWITCHMODE_SPEAKER
The device's speaker is active, the microphone is mute.

PHONEHOOKSWITCHMODE_MICSPEAKER
The device's microphone and speaker are both active.

PHONEHOOKSWITCHMODE_UNKNOWN
The device's hookswitch state is unknown or cannot be detected.

dwHandsetVolume

Specifies the current speaker volume of the phone's handset device. This is a value between 0x00000000 (silence) and 0x0000FFFF (maximum volume).

dwHandsetGain

Specifies the current microphone gain of the phone's handset device. This is a value between 0x00000000 (silence) and 0x0000FFFF (maximum gain).

dwSpeakerHookSwitchMode

Specifies the current hook switch mode of the phone's speakerphone. This field uses the following PHONEHOOKSWITCHMODE_ constants:

PHONEHOOKSWITCHMODE_ONHOOK
The device's microphone and speaker are both onhook.

PHONEHOOKSWITCHMODE_MIC
The devicer's microphone is active, the speaker is mute.

PHONEHOOKSWITCHMODE_SPEAKER
The device's speaker is active, the microphone is mute.

PHONEHOOKSWITCHMODE_MICSPEAKER
The device's microphone and speaker are both active.

PHONEHOOKSWITCHMODE_UNKNOWN
The device's hookswitch state is unknown or cannot be detected.

dwSpeakerVolume

Specifies the current speaker volume of the phone's speaker device. This is a value between 0x00000000 (silence) and 0x0000FFFF (maximum volume).

dwSpeakerGain

Specifies the current microphone gain of the phone's speaker device. This is a value between 0x00000000 (silence) and 0x0000FFFF (maximum gain).

dwHeadsetHookSwitchMode

Specifies the current hook switch mode of the phone's headset. This field uses the following PHONEHOOKSWITCHMODE_ constants:

PHONEHOOKSWITCHMODE_ONHOOK
The device's microphone and speaker are both onhook.

PHONEHOOKSWITCHMODE_MIC
The devicer's microphone is active, the speaker is mute.

PHONEHOOKSWITCHMODE_SPEAKER
The device's speaker is active, the microphone is mute.

PHONEHOOKSWITCHMODE_MICSPEAKER
The device's microphone and speaker are both active.

PHONEHOOKSWITCHMODE_UNKNOWN
The device's hookswitch state is unknown or cannot be detected.

dwHeadsetVolume
Specifies the current speaker volume of the phone's headset device. This is a value between 0x00000000 (silence) and 0x0000FFFF (maximum volume).

dwHeadsetGain
Specifies the current microphone gain of the phone's headset device. This is a value between 0x00000000 (silence) and 0x0000FFFF (maximum gain).

dwDisplaySize
dwDisplayOffset
The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized field containing the phone's current display information.

dwLampModesSize
dwLampModesOffset
The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized field containing the phone's current lamp modes.

dwOwnerNameSize
dwOwnerNameOffset
The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized field containing the name of the application that is the current owner of the phone device. The name is the application name provided by the application when it invoked **phoneInitialize**. If no application name was supplied, the application's module name is used instead. If the phone currently has no owner, **dwOwnerNameSize** is zero.

dwDevSpecificSize
dwDevSpecificOffset
The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device-specific field.

Device-specific extensions should use the DevSpecific (**dwDevSpecificSize** and **dwDevSpecificOffset**) variably sized area of this data structure.

VARSTRING

[New - Windows 95]

```
typedef struct varstring_tag {
  DWORD  dwTotalSize;
  DWORD  dwNeededSize;
  DWORD  dwUsedSize;

  DWORD  dwStringFormat;
  DWORD  dwStringSize;
  DWORD  dwStringOffset;
} VARSTRING, FAR *LPVARSTRING;
```

The **VARSTRING** structure is used for returning variably sized strings. It is used both by the line device class and the phone device class.

dwTotalSize

The total size in bytes allocated to this data structure.

dwNeededSize

The size in bytes for this data structure that is needed to hold all the returned information.

dwUsedSize

The size in bytes of the portion of this data structure that contains useful information.

dwStringFormat

Specifies the format of the string. This field uses the following STRINGFORMAT_ constants:

STRINGFORMAT_ASCII
: This is ASCII string format using one byte per character. The actual string is a NULL-terminated ASCII string with the terminating NULL accounted for in the string size. If **dwStringFormat** is STRINGFORMAT_ASCII, the actual string is a NULL-terminated ASCII string with the terminating NULL accounted for in the string size.

STRINGFORMAT_DBCS
: This is DBCS string format using one or two bytes per character.

STRINGFORMAT_UNICODE
: This is unicode string format using two bytes per character.

STRINGFORMAT_BINARY
: This is an array of unsigned characters. It could be used for numeric values.

dwStringSize
dwStringOffset

The size in bytes and the offset in bytes from the beginning of this data structure of the variably sized device field containing the string information.

Not extensible.

Phone Device Constants

Reference

Reference

PHONEBUTTONFUNCTION_ Constants

[New - Windows 95]

The PHONEBUTTONFUNCTION_ scalar constants describe the functions commonly assigned to buttons on telephone sets.

PHONEBUTTONFUNCTION_UNKNOWN
: A “dummy” function assignment that indicates that the exact function of the button is unknown or has not been assigned.

PHONEBUTTONFUNCTION_CONFERENCE
: Initiates a conference call or adds a call to a conference call.

PHONEBUTTONFUNCTION_TRANSFER
: Initiates a call transfer or completes the transfer of a call.

PHONEBUTTONFUNCTION_DROP
: Drops the active call.

PHONEBUTTONFUNCTION_HOLD
: Places the active call on hold.

PHONEBUTTONFUNCTION_RECALL
: Unholds a call.

PHONEBUTTONFUNCTION_DISCONNECT
: Disconnects a call, such as after initiating a transfer.

PHONEBUTTONFUNCTION_CONNECT
: Reconnects a call that is on consultation hold.

PHONEBUTTONFUNCTION_MSGWAITON
: Turns on a message waiting lamp.

PHONEBUTTONFUNCTION_MSGWAITOFF
: Turns off a message waiting lamp.

PHONEBUTTONFUNCTION_SELECTRING
: Allows the user to select the ring pattern of the phone.

PHONEBUTTONFUNCTION_ABBREVDIAL
: The number to dialed will be indicated using a short, abbreviated number consisting of one or a few digits.

PHONEBUTTONFUNCTION_FORWARD
: Initiates or changes call forwarding to this phone.

PHONEBUTTONFUNCTION_PICKUP
: Picks up a call ringing on another phone.

PHONEBUTTONFUNCTION_RINGAGAIN
: Initiates a request to be notified if a call cannot be completed normally because of a busy or no answer.

PHONEBUTTONFUNCTION_PARK
: Parks the active call on another phone, placing it on hold there.

PHONEBUTTONFUNCTION_REJECT
: Rejects an inbound call before the call has been answered.

PHONEBUTTONFUNCTION_REDIRECT
: Redirects an inbound call to another extension before the call has been answered.

PHONEBUTTONFUNCTION_MUTE
: Mutes the phone’s microphone device.

PHONEBUTTONFUNCTION_VOLUMEUP
: Increases the volume of audio through the phone’s handset speaker or speakerphone.

PHONEBUTTONFUNCTION_VOLUMEDOWN
Decreases the volume of audio through the phone’s handset speaker or speakerphone.
PHONEBUTTONFUNCTION_SPEAKERON
Turns the phone’s external speaker on.
PHONEBUTTONFUNCTION_SPEAKEROFF
Turns the phone’s external speaker off.
PHONEBUTTONFUNCTION_FLASH
Generates the equivalent of a on-hook/off-hook sequence. A flash typically indicates that any digits typed next are to be understood as commands to the switch. On many switches, places an active call on consultation hold.
PHONEBUTTONFUNCTION_DATAON
Indicates that the next call is a data call.
PHONEBUTTONFUNCTION_DATAOFF
Indicates that the next call is not a data call.
PHONEBUTTONFUNCTION_DONOTDISTURB
Places the phone in “do not disturb” mode; incoming calls receive a busy signal or are forwarded to an operator or voicemail system.
PHONEBUTTONFUNCTION_INTERCOM
Connect to the intercom to broadcast a page.
PHONEBUTTONFUNCTION_BRIDGEDAPP
Selects a particular appearance of a bridged address.
PHONEBUTTONFUNCTION_BUSY
Makes the phone appear “busy” to incoming calls.
PHONEBUTTONFUNCTION_CALLAPP
Selects a particular call appearance.
PHONEBUTTONFUNCTION_DATETIME
Causes the phone to display current date and time; this information would be sent by the switch.
PHONEBUTTONFUNCTION_DIRECTORY
Calls up directory service from the switch.
PHONEBUTTONFUNCTION_COVER
Forwards all calls destined for this phone to another phone used for coverage.
PHONEBUTTONFUNCTION_CALLID
Requests display of caller ID on the phone's display.
PHONEBUTTONFUNCTION_LASTNUM
Redials last number dialed.
PHONEBUTTONFUNCTION_NIGHTSRV
Places the phone in the mode it is configured for during night hours.
PHONEBUTTONFUNCTION_SENDCALLS
Sends all calls to another phone used for coverage; same as PHONEBUTTONFUNCTION_COVER.
PHONEBUTTONFUNCTION_MSGINDICATOR
Controls the message indicator lamp.
PHONEBUTTONFUNCTION_REPDIAL
Repertory dialing—the number to be dialed is provided as a shorthand following pressing of this button.
PHONEBUTTONFUNCTION_SETREPDIAL
Programs the shorthand-to-phone number mappings accessible by means of repertory dialing (the “REPDIAL” button).
PHONEBUTTONFUNCTION_SYSTEMSPEED
The number to be dialed is provided as a shorthand following pressing of this button. The mappings for system speed dialing are configured inside the switch.
PHONEBUTTONFUNCTION_STATIONSPEED

The number to be dialed is provided as a shorthand following pressing of this button. The mappings for station speed dialing are specific to this station (phone).

PHONEBUTTONFUNCTION_CAMPON

Camps-on an extension that returns a busy indication. When the remote station returns to idle, the phone will be rung with a distinctive patterns. Picking up the local phone re-initiates the call.

PHONEBUTTONFUNCTION_SAVEREPEAT

When pressed while a call or call attempt is active, it will remember that call's number or command. When pressed while no call is active (such as during dialtone), it repeats the most saved command.

PHONEBUTTONFUNCTION_QUEUECALL

Queues a call to an outside number after it encounters a trunk-busy indiction. When a trunk becomes later available, the phone will be rung with a distinctive pattern. Picking up the local phone re-initiates the call.

PHONEBUTTONFUNCTION_NONE

A “dummy” function assignment that indicates that the button does not have a function.

Values in the range 0x80000000 to 0xFFFFFFFF can be assigned for device-specific extensions; values in the range 0x00000000 to 0x7FFFFFFF are reserved.

The PHONEBUTTONFUNCTION_ constants have values commonly found on current telephone sets. These button functions can be used to invoke the corresponding function from the switch using **TSPI_lineDevSpecificFeature**. Note that TAPI.DLL does not define the semantics of the button functions, it only provides access to the corresponding function. The behavior associated with each of the function values above is generic and may vary based on the telephony environment.

PHONEBUTTONMODE_ Constants

[New - Windows 95]

The PHONEBUTTONMODE_ bit-flag constants describe the button classes.

PHONEBUTTONMODE_DUMMY

This value is used to describe a button/lamp position that has no corresponding button, but has only a lamp.

PHONEBUTTONMODE_CALL

The button is assigned to a call appearance.

PHONEBUTTONMODE_FEATURE

The button is assigned to requesting features from the switch, such as hold, conference, and transfer.

PHONEBUTTONMODE_KEYPAD

The button is one of the twelve keypad buttons, ‘0’ through ‘9’, ‘*’, and ‘#’.

PHONEBUTTONMODE_LOCAL

The button is a local function button, such as mute or volume control.

PHONEBUTTONMODE_DISPLAY

The button is a “soft” button associated with the phone’s display. A phone set can have zero or more display buttons.

Not extensible. All 32 bits are reserved.

This enumeration type is used in the **PHONECAPS** data structure to describe the meaning associated with the phone’s buttons.

PHONEBUTTONSTATE_ Constants

[New - Windows 95]

The PHONEBUTTONSTATE_ bit-flag constants describe various button positions.

PHONEBUTTONSTATE_UP

The button is in the “up” state.

PHONEBUTTONSTATE_DOWN

The button is in the “down” state (pressed down).

Not extensible. All 32 bits are reserved.

PHONEERR_ Constants

[New - Windows 95]

This is the list of error codes that the implementation may return when invoking operations on phone devices. Consult the individual function descriptions to determine which of these error codes each function may return.

PHONEERR_ALLOCATED
The specified resource is already allocated.

PHONEERR_BADDEVICEID
The specified phone device ID is invalid or out of range.

PHONEERR_INCOMPATIBLEAPIVERSION
The passed TSPI version did not match an interface version definition supported by the service provider.

PHONEERR_INCOMPATIBLEEXTVERSION
The caller requested an extension version or a version range that cannot be supported by the service provider.

PHONEERR_INIFILECORRUPT
The TELEPHON.INI file was corrupted.

PHONEERR_INUSE
The device is currently in use, or the phone device is in use and cannot currently be configured.

PHONEERR_INVALBUTTONLAMPID
The specified Button ID is out of range, invalid, or not supported by the service provider or phone.

PHONEERR_INVALBUTTONMODE
The button mode parameter is invalid.

PHONEERR_INVALBUTTONSTATE
The button states parameter is invalid.

PHONEERR_INVALDATAID
The specified data ID is invalid or the number of bytes specified in *dwSize* cannot be read from that location.

PHONEERR_INVALDEVICECLASS
The service provider or phone does not support the indicated device class.

PHONEERR_INVALHOOKSWITCHDEV
The hookswitch device parameter is invalid or not supported by the phone or service provider.

PHONEERR_INVALLAMPMODE
The specified lamp mode parameter is invalid or not supported by the phone or service provider.

PHONEERR_INVALPARAM
The row or column parameters or window handle parameter are invalid or out of range.

PHONEERR_INVALPHONEHANDLE
The specified device handle, such as *hdPhone*, is invalid.

PHONEERR_INVALPHONESTATE
The phone device is not in a valid state for the requested operation, or the specified phone states parameter is invalid.

PHONEERR_INVALPOINTER
The specified pointer parameter is invalid.

PHONEERR_INVALRINGMODE
The ring mode parameter is invalid or not supported by the phone or service provider.

PHONEERR_NODRIVER
This error may be returned from a function when the driver finds that one of its components is

corrupt in a way that was not detected at initialization time. For example, if a service provider consists of multiple components such as a Virtual Device Driver (VxD) in addition to the usual DLL (exporting the TSPI interface and having the ".TSP" extension), PHONEERR_NODRIVER could be returned to indicate that the VxD component was missing.

PHONEERR_NOMEM
Unable to allocate or lock memory.

PHONEERR_OPERATIONFAILED
The operation failed for unspecified reasons.

PHONEERR_OPERATIONUNAVAIL
The operation is not available.

PHONEERR_RESOURCEUNAVAIL
There are not enough resources available to complete the request.

The values 0xC0000000 through 0xFFFFFFFF are available for device-specific extensions, the values 0x80000000 through 0xBFFFFFFF are reserved, and 0x00000000 through 0x7FFFFFFF are used as request IDs.

PHONEHOOKSWITCHDEV_ Constants

[New - Windows 95]

The PHONEHOOKSWITCHDEV_ bit-flag constants describe various audio I/O devices each with its own hookswitch controllable from the computer.

PHONEHOOKSWITCHDEV_HANDSET
This is the ubiquitous, hand held ear and mouth piece.

PHONEHOOKSWITCHDEV_SPEAKER
This is a built-in loudspeaker and microphone. This could also be an externally connected adjunct speaker to the telephone set.

PHONEHOOKSWITCHDEV_HEADSET
This is a headset connected to the phone set.

Not extensible. All 32 bits are reserved.

These constants are used in the **PHONECAPS** data structure to indicate the hookswitch device capabilities of a phone device. The **PHONESTATUS** structure reports the state of the phone's hookswitch devices. The function **TSPI_phoneSetHookSwitch** and **TSPI_phoneGetHookSwitch** use it as a parameter to select the phone's I/O device.

PHONEHOOKSWITCHMODE_ Constants

[New - Windows 95]

The PHONEHOOKSWITCHMODE_ bit-flag constants describe the microphone and speaker components of a hookswitch device.

PHONEHOOKSWITCHMODE_ONHOOK
The device's microphone and speaker are both onhook.

PHONEHOOKSWITCHMODE_MIC
The devicer's microphone is active, the speaker is mute.

PHONEHOOKSWITCHMODE_SPEAKER
The device's speaker is active, the microphone is mute.

PHONEHOOKSWITCHMODE_MICSPEAKER
The device's microphone and speaker are both active.

PHONEHOOKSWITCHMODE_UNKNOWN
The device's hookswitch mode is currently unknown.

Not extensible. All 32 bits are reserved.

These constants are used to provide individual level of control over the microphone and speaker components of a phone device.

PHONELAMPMODE_ Constants

[New - Windows 95]

The PHONELAMPMODE_ bit-flag constants describe various ways in which a phone's lamp can be lit.

PHONELAMPMODE_DUMMY
This value is used to describe a button/lamp position that has no corresponding lamp.

PHONELAMPMODE_OFF
The lamp is off.

PHONELAMPMODE_STEADY
Steady means the lamp is continuously lit.

PHONELAMPMODE_WINK
Wink means normal rate on and off.

PHONELAMPMODE_FLASH
Flash means slow on and off.

PHONELAMPMODE_FLUTTER
Flutter means fast on and off.

PHONELAMPMODE_BROKENFLUTTER
Broken flutter is the superposition of flash and flutter.

PHONELAMPMODE_UNKNOWN
The lamp mode is currently unknown.

The high-order 16 bits can be assigned for device-specific extensions. The low-order 16 bits are reserved.

Where the exact on and off cadences may differ across phone sets from different vendors, mapping of actual lamp lighting patterns for most phones onto the values listed above should be straightforward.

PHONESTATE_ Constants

[New - Windows 95]

The PHONESTATE_ bit-flag constants describe various status items for a phone device.

PHONESTATE_OTHER
A phone status information item other than those listed below has changed.

PHONESTATE_CONNECTED
The connection between the phone device and TAPI.DLL was just made. This happens when TAPI.DLL is first invoked, or when the wire connecting the phone to the computer is plugged in with TAPI.DLL active.

PHONESTATE_DISCONNECTED
The connection between the phone device and TAPI.DLL was just broken. This happens when the wire connecting the phone set to the computer is unplugged while TAPI.DLL is active.

PHONESTATE_OWNER
The number of owners for the phone device.

PHONESTATE_MONITORS
The number of monitors for the phone device.

PHONESTATE_DISPLAY
The display of the phone has changed.

PHONESTATE_LAMP
A lamp of the phone has changed.

PHONESTATE_RINGMODE
The ring mode of the phone has changed.

PHONESTATE_RINGVOLUME
The ring volume of the phone has changed.

PHONESTATE_HANDSETHOOKSWITCH

The handset hookswitch state has changed.

PHONESTATE_HANDSETVOLUME
The handset's speaker volume setting has changed.

PHONESTATE_HANDSETGAIN
The handset's microphone gain setting has changed.

PHONESTATE_SPEAKERHOOKSWITCH
The speakerphone's hookswitch state has changed.

PHONESTATE_SPEAKERVOLUME
The speakerphone's speaker volume setting has changed.

PHONESTATE_SPEAKERGAIN
The speakerphone's microphone gain setting state has changed.

PHONESTATE_HEADSETHOOKSWITCH
The headset's hookswitch state has changed.

PHONESTATE_HEADSETVOLUME
The headset's speaker volume setting has changed.

PHONESTATE_HEADSETGAIN
The headset's microphone gain setting has changed.

PHONESTATE_SUSPEND
The application's use of the phone is temporarily suspended.

PHONESTATE_RESUME
The application's use of the phone device is resumed after having been suspended for some time.

PHONESTATE_DEVSPECIFIC
The phone's device-specific information has changed.

PHONESTATE_REINIT
Items have changed in the configuration of phone devices. To become aware of these changes (as for the appearance of new phone devices) the application should reinitialize its use of TAPI.

Not extensible. All 32 bits are reserved.

PHONESTATUSFLAGS_ Constants

[New - Windows 95]

The PHONESTATUSFLAGS_ bit-flag constants describe phone device status information.

PHONESTATUSFLAGS_CONNECTED
Specifies whether the phone is currently connected to TAPI. TRUE if connected, FALSE otherwise.

PHONESTATUSFLAGS_SUSPENDED
Specifies whether TAPI's manipulation of the phone device is suspended. TRUE if suspended, FALSE otherwise. An application's use of a phone device may be temporarily suspended when the switch wants to manipulate the phone in a way that cannot tolerate interference from the application.

Not extensible. All 32 bits are reserved.

STRINGFORMAT_ Constants

[New - Windows 95]

The STRINGFORMAT_ enumeration constants describe different string formats.

STRINGFORMAT_ASCII
Specifies standard ASCII character format using one byte per character.

STRINGFORMAT_DBCS
Specifies standard DBCS character format using two bytes per character.

STRINGFORMAT_UNICODE
Specifies standard Unicode character format using two bytes per character.

STRINGFORMAT_BINARY

This is an array of unsigned characters; could be used for numeric values.
Not extensible. All 32 bits are reserved.

Configuration Information

About Configuration Information

Configuration Information

About Configuration Information

Introduction

This appendix specifies the configuration parameters associated with Windows Telephony. These parameters are manipulated by Control Panel applets, Telephony service providers, and Telephony applications. Much of the configuration information related to TAPI, such as the default priorities users wish to assign to applications, is stored in the TELEPHON.INI file. A small amount of information is stored in the file WIN.INI. (Priorities are used to determine which of two or more applications capable of accepting an incoming call of a given media mode (such as "datamodem") actually receives the call. Telephony applications should offer a Preferences option that, according to the application design, can let users reset the current application's priority. These settings remain constant in TELEPHON.INI until changed by a later edit.)

The only portions of TELEPHON.INI that are of interest to service provider programmers are the [`Providers`] and [`Provider<PPID>`] sections. All other sections of TELEPHON.INI are to be modified only by applications or Telephony Control Panel applets, and should not be modified in any way by service providers.

WIN.INI

Windows Telephony parameters are stored in TELEPHON.INI. TAPI.DLL must be informed when any of the TELEPHON.INI parameters defined in this chapter have been changed. This is done through a special section in WIN.INI. This section consists of a single entry:

```
[Windows Telephony]
TelephonINIChanged=yyyy-mm-dd hh:mm:ss
```

For example,

```
TelephonINIChanged=1993-06-03 16:48:56
```

would indicate that TELEPHON.INI was last updated on June 3, 1993, at 4:48:56 p.m. Whenever any piece of software such as an application or a service provider modifies any of the defined parameters in TELEPHON.INI, it must update this entry in WIN.INI and broadcast the WM_WININICHANGE message defined by Windows to all top-level applications. This informs TAPI.DLL that parameters have changed. TAPI.DLL, in turn takes care of notifying all active telephony applications through the **LINEDEVSTATE_REINIT** or **PHONESTATE_REINIT** messages, and notifying service providers by shutting them down and reinitializing them (**TSPI_providerShutdown, TSPI_providerInit**). Neither applications nor service providers need to attempt to detect TELEPHON.INI changes directly.

Providers

The “[Providers]” section of TELEPHON.INI specifies the permanent provider IDs and filenames of installed Telephony service providers. This information is used so that TAPI.DLL can identify and load each provider.

This section uses the following format:

```
[Providers]
NumProviders=<count>
NextProviderID=<nextPPID>
ProviderID<index>=<PPID>
ProviderFilename<index>=<filename>
```

For example:

```
[Providers]
NumProviders=2
NextProviderID=5
ProviderID0=3
ProviderFilename0=UNIMODEM.TSP
ProviderID1=4
ProviderFilename1=PDI-1000.TSP
```

The NumProviders entry indicates how many service providers are installed (and how many ProviderID<index> and ProviderFilename<index> entries appear in the section). The NextProviderID entry indicates the Permanent Provider ID (<nextPPID>) that will be assigned to the next provider to be installed. Both parameters are set to 0 when Telephony is installed; the values are subsequently controlled by any software that installs or removes service providers, such as the Windows Telephony Control Panel.

The ProviderID<index> entry (one per provider) assigns a permanent <PPID>to a Telephony service provider; this ID is used to link parameters in other sections and files to the provider information. These entries are created by any software that installs or removes service providers, such as the Windows Telephony Control Panel.

The ProviderFilename<index> (one per provider) identifies the <filename>(in \WINDOWS\SYSTEM) of the service provider DLL. Telephony service providers should use ".TSP" as a file extension to allow the Telephony Control Panel applet to distinguish them from other DLLs. These entries are created by any software that installs or removes service providers, such as the Telephony Control Panel.

In both the ProviderID<index> and ProviderFilename<index> entries, <index> takes values from 0 to one less than <count> (as specified in NumProviders). These entries must be renumbered as providers are deleted, but the permanent provider IDs should not be changed. Although all these entries must be present, there is no requirement that they appear in numerical order.

Adding a New Service Provider

When adding a new provider the following rules must be observed:

There can be no more than 32767 providers; this is the maximum value permitted for <count>.

All providers must have unique <PPID> values. <nextPPID> is a "hint" indicating a likely <PPID> value for a new provider. It should be incremented and wrapped around after 32767 if necessary if it is used for a new provider.

The ProviderID<index> and ProviderFilename<index> entries for the new provider must be created before calling the service provider.

The provider's **TSPI_providerInstall** procedure can be called to create and initialize the [Provider<PPID>] section described below. The Control Panel or whichever program called **TSPI_providerInstall** has the responsibility to update TelephonINIChanged and issue a WM_WININICHANGE message.

Similarly, when removing a service provider, several rules must be followed:

The **TSPI_providerRemove** procedure may be called to remove the [Provider<PPID>] section described below. The control panel—or whichever program called **TSPI_providerRemove**—has the responsibility to update TelephonINIChanged and issue a WM_WININICHANGE message.

The ProviderID<index> and ProviderFilename<index> entries for the provider must be removed after the [Provider<PPID>] section is removed.

Remaining ProviderID<index> and ProviderFilename<index> entries with higher <index> values than the one being removed must be renumbered to maintain a continguous range of indexes.

The <count> value must be decremented to reflect the new number of providers.

Windows Telephony performs a number of consistency checks on the TELEPHON.INI file and the configured service providers when telephony operations are started. There is an instance in which these consistency requirements are relaxed: If a service provider is defined in the [Providers] section but its <filename> cannot be loaded, that service provider is silently ignored as if it were not even present in the list of providers. However, if the service provider loads, full initialization of that provider must succeed otherwise an "initialization file corrupt" error is reported to applications.

Provider<PPID> (Provider-specific Sections)

For each telephony service provider defined in the [Providers] section, there must be a [Provider<PPID>] section. The <PPID> value is the Permanent Provider ID defined for that service provider in the corresponding ProviderID<index>=<PPID> entry in the [Providers] section. For example, corresponding to an entry such as "ProviderID2=278" there would be a section named "[Provider278]".

It is the responsibility of the service provider to create its provider-specific section in TELEPHON.INI at the time the **TSPI_providerInstall** function is called. It is also the responsibility of the service provider to delete its provider-specific section in TELEPHON.INI at the time the **TSPI_providerRemove** function is called. The service provider may update the information in the section whenever **TSPI_providerConfig**, **TSPI_lineConfigDialog**, or **TSPI_phoneConfigDialog** is called, or at other times when appropriate. Only the service provider writes entries in its provider-specific section; neither TAPI.DLL nor any other Windows Telephony component writes any entries to provider-specific section. TAPI.DLL reads only two entries in the provider-specific section—NumLines and NumPhones (defined below)—and all other entries in the section are the responsibility of the provider.

Every provider-specific section in TELEPHON.INI specifies, at a minimum, the number of lines and phones supported (or currently configured, for providers which support multiple devices) by that provider. This section consists, at a minimum, of the following:

```
[Provider<PPID>]
NumLines=<count>
NumPhones=<count>
```

For example:

```
[Provider3]
NumLines=1
NumPhones=0
```

The **TSPI_providerInstall** function supplied by a service provider must create at least the entries listed above. The service provider may include any other entries in its section which are necessary to fulfill the responsibilities of the provider as specified in the Windows Telephony Service Provider Interface Specification, or for other private configuration purposes. In particular, it is the responsibility of the service provider to provide unique device IDs for every line and phone device, which are valid over a long span of time.

Service providers are responsible for forming their own 32-bit permanent device IDs for each line and phone device they support and returning them through the appropriate TSPI function. These IDs must meet the following two requirements. (1) The most-significant 16 bits of the ID is the service provider's <PPID>. (2) The least-significant 16 bits of the ID is unique within the set of line IDs or phone IDs supported by that service provider. Together, these requirements guarantee that each line's permanent ID is unique across the total set of lines provided by all service providers and that each phone's permanent ID is unique across the total set of phones.

The suggested mechanism to accomplish this is to define the following entries in the [Provider<PPID>] section:

```
NextLineID=<PLID>
NextPhoneID=<PFID>
LineID<index>=<PLID>
PhoneID<index>=<PFID>
```

Then a service provider can assign unique permanent IDs to a line conveniently using the following formula:

```
permanentLineID = (PPID << 16) + PLID
```

A similar formula assigns permanent IDs to phones.

If no Phone devices are supported by the provider, then only the “Line” entries need be included, and vice-versa (however, NumLines and NumPhones must always be included). NextLineID and

NextPhoneID would be used (like NextProviderID) to indicate the next permanent ID to be used (in an increasing fashion to minimize reusing of ID values).

The LineID<index> and PhoneID<index> entries provide convenient, permanent "partial ID" tags that are independent of the permanent Provider ID assigned. If this scheme of partial ID tags is used, it is suggested that other device-specific parameters (such as the “friendly name” of a device) be linked on the basis of the permanent ID tag; this way, it is not necessary to renumber those other parameters if a line or phone is deleted. For example, if the third line device on a provider has permament line ID 257, and has a couple of additional parameters, they could be stored in TELEPHON.INI as follows:

```
LineID3=257
LineName257="Microsoft ATSP"
LinePortDevice257="COM4"
LineInitString257="ATM2L0X0"
```

Of course, this tagging of additional parameters is not necessary if the provider supports one, and only one, device. In that case, only the NumLines and NumPhones entries need be included; other parameters can have arbitrary, non-indexed names. Also, providers may choose to use an entirely different mechanism for assigning permanent device IDs, as long as the requirements that the IDs be unique and contain the permanent provider ID in their high-order word are obeyed.

Handoff Priorities

The “[HandoffPriorities]” section of TELEPHON.INI specifies the priority of applications to handle incoming calls and call handoffs of particular media types and for Assisted Telephony requests. It can be modified only by the writer of the telephonic application.

When a call is initially received or handed off, TAPI.DLL checks the applications that have the line open to determine which of them expressed interest in calls of the current media mode (specified by the provider or application handing off the call). If more than one application has the line open and is interested in calls of the current media mode, TAPI.DLL checks the priorities specified in this section to determine which application will get the call. For each media type, applications are listed in the “[HandoffPriorities]” section of TELEPHON.INI in order of preference (earlier ones in the list are preferred over later ones). In other words, TAPI attempts first to choose as the target of the handoff the first listed application that has the line open for that media type.

Applications can also be targets of handoffs without being listed. Among unlisted applications, the order of preference is arbitrary, but listed applications are preferred to unlisted applications. There is no direct association between the media modes for which applications are registered and particular media stream APIs; it is up to the application to know which API is used to access a particular media stream.

Likewise, when a Assisted Telephony request is received, TAPI.DLL checks the applications that have registered to process that type of request. If more than one application has has registered for that type of request, TAPI.DLL checks the priorities specified in this section to determine which application will be signalled to process the request (with the highest priority being assigned to the first listed application under that request type, and proceeding in the order listed). If none of the open, interested applications are listed in the corresponding request entry in this section (or if there is no entry for that request type), the application to receive the request indication is arbitrary.

If no application is registered to handle the request type (for a **tapiRequestMakeCall**), TAPI.DLL attempts to launch the applications listed in this section, in order, until one of them successfully starts up. The request then remains pending until an application (perhaps the one that launched) registers to receive it.

Entries in the HandoffPriorities Section

Users may manipulate the entries in [HandoffPriorities] directly, although it's better that they use the Preferences settings of Telephony-aware applications. Such applications should be capable of adding entries to [HandoffPriorities] for all the media modes and request types they support, and of rearranging the application names in entries in the section (to set themselves as the highest priority, for example). Entries in the [HandoffPriorities] section are not manipulated through the Telephony control panel.

The [HandoffPriorities] section has the following format:

```
[HandoffPriorities]
RequestMakeCall=[<appname>[,<appname>]...]
RequestMediaCall=[<appname>[,<appname>]...]
unknown=[<appname>[,<appname>]...]
interactivevoice=[<appname>[,<appname>]...]
automatedvoice=[<appname>[,<appname>]...]
g3fax=[<appname>[,<appname>]...]
g4fax=[<appname>[,<appname>]...]
datamodem=[<appname>[,<appname>]...]
teletex=[<appname>[,<appname>]...]
videotex=[<appname>[,<appname>]...]
telex=[<appname>[,<appname>]...]
mixed=[<appname>[,<appname>]...]
tdd=[<appname>[,<appname>]...]
adsi=[<appname>[,<appname>]...]
digitaldata=[<appname>[,<appname>]...]
NumExtends=<count>
extend<index>=<extid0>,<extid1>,<extid2>,<extid3>,<bit>,"<friendlyname>"
    [,<appname>[,<appname>]...]
```

where each <appname> is the module filename of a Windows Telephony application.

The first two entries correspond to the request types defined in LINEREQUESTMODE (LINEREQUESTMODE_DROP requests are automatically routed to the same application which handles LINEREQUESTMODE_MEDIACALL requests, since only media calls can be dropped in Assisted Telephony; therefore, the same priorities apply to both).

The entries that follow correspond to the standard LINEMEDIAMODEs defined in the Telephony API Specification.

NumExtends provides a count of the number of extend<index> lines which follow (with <index> taking values from 0 through <count>-1). By default, it is set to 0. It may be adjusted by any application which adds or deletes an extended media mode.

Each extend<index> line gives the four DWORDs of the Extension Identifier for the extension set, a bit identifier corresponding to the LINEMEDIAMODE bit assigned in that extension set, a friendly name for the media mode (assigned by the provider or application, based on *a priori* knowledge), and the application priorities.

> **Note** The only extended media modes that applications are permitted to use are the ones defined in TELEPHON.INI. The attempt to use an extended media mode outside this set returns a **LINEERR_INVALMEDIAMODE** error to the application, even if the service provider for that line would otherwise support that extended media mode.

<extid0>, <extid1>, <extid2>, and <extid3> shall each be expressed as exactly eight ASCII hex digits. For example, if the EXTIDGEN program outputs an extension ID as:

```
dwExtensionID0 = 0xA2A9E740;
dwExtensionID1 = 0xA0A51068;
dwExtensionID2 = 0x968A0800;
dwExtensionID3 = 0x2B312E8A;
```

the extension ID would be specified in an extend<index> entry as:

```
extend<index>=A2A9E740,A0A51068,968A0800,2B312E8A,...
```

<bit> shall be a single decimal digit with a value from 1 to 8. These identifiers correspond to the following bit values in LINEMEDIAMODE (which are the only values available for assignment to proprietary media modes in provider extensions):

```
1   0x80000000
2   0x40000000
3   0x20000000
4   0x10000000
5   0x08000000
6   0x04000000
7   0x02000000
8   0x01000000
```

The following steps are used to add a new extend<index> entry:

1. If <count> is already equal to 32767, no more extend<index> can be added.
2. Obtain <extid0>, <extid1>, <extid2>, <extid3>, <bit>, and <friendlyname> from *a priori* knowledge.
3. Add an extend<index> entry to the section, with <index> set to the value of <count>.
4. Increment <count> by 1. Update the extends entry to show the new value of <count>.
5. Update TelephonINIChanged, and issue a WM_WININICHANGE message.

The following steps are used to delete an extend<index> entry:

1. Scan the extend<index> entries to find the entry to be deleted. If it is the last extend<index> (<index> is equal to <count>-1), skip step 2.
2. Copy the information for all higher-numbered extend<index> entries to one lower value of <index> (for example, copy extend6 to extend5). This will leave the highest-numbered extend<index> entry duplicating the next to highest-numbered extend<index> entry.
3. Delete the highest numbered extend<index> entry (<index> equal to <count>-1)
4. Decrement <count> by 1.
5. Update the extends entry to show the new value of <count>.
6. Update TelephonINIChanged, and issue a WM_WININICHANGE message.

Note that duplicate entries in the [HandoffPriorities] section are not allowed. For example, if two (or more) entries exist in TELEPHON.INI such as

```
G3fax=<some stuff>
G3fax=<more stuff>
```

or,

```
extension<index>=<extension ID>,bit, <some stuff>
extensiony=<extension ID>,bit, <more stuff>
```

actual application priorities for these media modes are arbitrary.

Also note that leading "0x" as in "0x <extensionID> is not allowed in the TELEPHON.INI file format.

Locations

The “[Locations]” section of TELEPHON.INI specifies the locations that are defined by the user. A Location includes a user-defined name, the prefixes necessary to obtain access to an “outside line” at the location, the country code and area/city code of the location, the preferred dialing method for calls placed at the location, and additional information about dialing calls. This information is maintained by the Telephony control panel application.

This section has the following format:

```
[Locations]
Locations=<count>,<nextID>
Location<index>=<id>,"<friendlyname>","<localprefix>","<LDprefix>",
"<areacode>",<countrycode>,<preferredcardid>,<hint>,
    <insertareacode>,"<tollprefixes>"
CurrentLocation=<id>,<hint>
```

For example:

```
[Locations]
Locations=3,458
Location0=0,"Default Location","","","206",1,0,0,1,""
Location1=233,"Seattle Office","9,","9,","206",1,0,0,
    1,"223,224,233,281,282,283,284,285,286,287,..."
Location2=457,"Boston Office","9","8","617",1,2,2,1,
    ""
CurrentLocation=233,1
```

The Locations entry indicates how many locations are defined (and how many Location<index> entries appear in the section) with the <count> parameter, and the next permanent location ID to be assigned with the <nextID> parameter. Both parameters are set to 0 when Telephony is installed; the value is subsequently controlled by the Locations function of the control panel.

Each Location<index> entry identifies a location, and gives its permanent ID (used to link parameters in other files to the location). "<index>" takes values from 0 to one less than <count>. These entries may be renumbered as locations are deleted. Location0 is the "Default Location", and can never be removed; its <id> is always 0, its name should not be changed, but all of the other parameters may be changed by the user.

Within each Location<index> entry, the individual fields have the following meanings:

<id> is the permanent ID for the location, which is used to link parameters to other files to the location. Once a permanent ID is assigned to a location, it is never changed. These IDs shall be in the range 0 to 32767 (when 32767 is used, <nextID> shall wrap to 0).

"<friendlyname>" is the user-assigned name for the location, entered through the Telephony control panel. It must be enclosed in quotation marks, and must not contain quotation marks.

"<localprefix>" is the dialable string to be dialed before all calls determined to be “local” (same country, same area code, not in the toll list).

"<LDprefix>" is the dialable string to be dialed before all calls that are not local (different country, different area code, or in the toll list).

"<areacode>" is the area/city code associated with the current location. If this is empty, the current country is presumed to have a flat numbering system without city codes (all local numbers unique throughout the country). This is used in determining whether the canonical number is local or long distance.

<countrycode> is the country code associated with the current location. This is linked to an entry in the "[Countries]" section to determine the dialing procedures when the selected dialing method is “Direct” (CallingCard permanent ID 0); if the code does not match an entry in the [Countries] section, then canonical translation fails and the number is returned unmodified. It is also used in determining whether the canonical number is local or long distance. This value is returned with the **lineTranslateAddress** function of the Telephony API.

<preferredcardid> is the preferred dialing method which will be preselected when the user requests a dialing operation. It must match one of the permanent IDs of an entry in the [Cards] section; if it does not, it is presumed to be 0 (“Direct”).

<hint> indicates the number of the Card<index> entry which matched <preferredcardid> at the time this Location<index> was last modified; to save time, the DLL can check this entry first when trying to find the information on the Card<index>, before scanning all locations (which should only be necessary after a card entry is removed).

<insertareacode> shall have a value of 0 or 1. 0 indicates that toll calls should not include the area code before the local number; 1 indicates that toll calls should have the area code before the local number. This is applicable only when <countrycode> is 1 (North America), the country code and area code being dialed are the same as for the current location, and the prefix of the local number appears in <tollprefixes>.

"<tollprefixes>" is a list of three-digit prefixes for calls which receive special treatment. If the country code for the current location is 1, the country code and area code being dialed are the same as for the current location, and the prefix of the local number matches an entry in this list, then the number will be dialed according to the <longdistance> dialing rule rather than the <samearea> dialing rule. The inclusion of the area code as part of the local number can be controlled by the <insertareacode> parameter. The entries within the list consist of three-digit numbers, separated by comma characters (0x2C), preferably sorted in ascending numeric order.

The CurrentLocation entry gives the <id> of the location at which the user is presently located. This may be set manually by the user through the Telephony control panel or a prompt at boot time, automatically by a boot-time process which examines physical characteristics of the system (such as whether or not a notebook computer is in a docking station), or through the **lineTranslateAddress** function in the Telephony API. The value should match an <id> in an existing Location<index> entry; if it does not, the current location should be assumed to be the default location (permanent ID 0). CurrentLocation should be set to "0,0" when TELEPHON.INI is first created. The <hint> parameter indicates the number of the Location<index> entry which matched <id> at the time CurrentLocation was last set; to save time, the DLL can check this entry first when trying to find the information on the CurrentLocation, before scanning all locations (which should only be necessary after a location is removed).

Calling Cards

The “[Cards]” section of TELEPHON.INI specifies information related to a method of dialing telephone calls. This method may, or may not, include an associated telephone calling card number (the dialing methods are referring to generically as “calling cards” because use of a calling card is the most frequent reason for using an alternative dialing procedure). The calling card numbers are scrambled to provide security against non-determined, casual observation (a person looking at the TELEPHON.INI file with a text editor); greater security will depend on system-level file or system access security (as in the case in which a notebook computer is stolen).

This section will have the following format:

```
[Cards]
Cards=<count>,<nextID>
Card<index>=<id>,"<friendlyname>","<scrambledcardnum>",
    "<samearea>","<longdistance>","<international>",<hidden>
```

For example:

```
[Cards]
Cards=5,4
Card0=0,"None (Direct Dial)","","","","",0
Card1=1,"AT&T via 10288","438 293 482 1938",
    "102880FG$H","102880FG$H","1028801EFG$H",0
Card2=2,"AT&T via 1-800-321-0288","",
    "18003210288$FG$H","18003210288$FG$H",
    "18003210288$01EFG$H",1
Card3=3,"Direct With Security Code",
    "","G","1FG,3948","011EFG,3948",0
Card4=4,"USA Direct from Italy","549 304 593 2049",
    "","","P1721011$TFG$H",0
```

The Cards entry indicates how many locations are defined (and how many Card<index> entries appear in the section) with the <count> parameter, and the next permanent card ID to be assigned with the <nextID> parameter. The values of these parameters are controlled by the Telephony control panel. Their initial settings when Telephony is installed is dependent on the number of predefined Card<index> entries shipped with the product (which is still to be determined).

Each Card<index> entry identifies a card, giving its permanent ID (used to link parameters in other files to the card). "<index>" takes values from 0 to one less than <count>. These entries may be renumbered as cards are deleted. The process for adding and deleting cards is similar to that for drivers, and is not repeated here. Card0 shall not be removable or modifiable by the user; it is a placeholder used to indicate that the direct dialing rules for the country specified in CurrentLocation should be used for dialing.

The parameters included in each Cardx entry are as follows:

<id> is the permanent ID associated with this card, assigned by the control panel. Once assigned, it is never changed. It is used to link the <preferredcardid> for a Location<index> entry to the associated card in a manner insensitive to the insertion and remove of Card<index> entries. These IDs shall always be in the range of 0 to 32767.

"<friendlyname>" is the user-assigned name for the card. It must be enclosed in quotation marks and may not contain quotation marks.

"<scrambledcardnum>" is the card number associated with this entry (if any; it may be "" for a Card<index> entry which actually specifies alternative direct dial rules). The contents are scrambled in order to provide security from accidental viewing.

"<samearea>", "<longdistance>", and "<international>" are the dialable string templates for the three classes of calls (same area code, different area code but same country, and different country code, respectively). The contents of these strings should be selected from the following list (spaces, hyphens, periods, and parentheses may be included for formatting purposes):

0-9 Dialable digits

A-D	Dialable digits
#	Dial DTMF "#"
*	Dial DTMF "*"
T	DTMF dial subsequent digits
P	Pulse dial subsequent digits
!	Hook flash
,	Pause during dialing
W	Wait for second dial tone
@	Wait for quiet answer
$	Wait for "bong" tone
?	Prompt user before proceeding
E	Insert country code
F	Insert area code
G	Insert subscriber number
H	Insert calling card number
I	Insert area code (area codes are optional; used in country entries only)

<hidden> determines whether or not this entry should be displayed in lists of available dialing methods displayed to the user by applications (they are delivered to applications by the **lineGetTranslateCaps** function). 0 indicates that the entry should always be included, and 1 indicates that it should be hidden (not included) ***if the <scrambledcardnum> string is empty*** (""). The function **lineGetTranslateCaps** will not return to the application entries for calling cards which have this parameter set to 1 and which have a null card number field.

Countries

The “[Countries]” section of TELEPHON.INI specifies the well-known direct dialing procedures for calls placed within specific countries. The contents of this section are shipped with Windows Telephony, are not intended to be edited by the user, and could, in future releases, be moved to a compressed binary file for quick access; however, the information is contained in the TELEPHON.INI file, at least in the initial release, to facilitate user editing (under the guidance of Microsoft Product Support Services) if problems are discovered.

This section will have the following format:

```
[Countries]
Country<index>=<countrycode>,<nextcountry>,"<name>","<samearea>",
    "<longdistance>","<international>"
```

For example:

```
[Countries]
Country1=1,100,"North America and Caribbean","G","1FG","011EFG"
Country100=1,101,"United States of America"
Country101=1,102,"Anguilla"
Country102=1,103,"Antigua"
Country103=1,104,"Bahamas"
Country104=1,105,"Barbados"
Country105=1,106,"Bermuda"
...
Country98=98,0,"Iran","G","0FG","00EFG"
```

There are two kinds of entries in the [Countries] section. The first, known as “primary” entries, are indicated by the <countrycode> parameter equaling the value of <index>. Primary entries specify the name of a World Numbering Plan Area as specified in CCITT Recommendation E.163, and include the direct dialing rules applicable to that numbering plan area. The second type of entries, known as “secondary” entries, are indicated by the <countrycode> parameter not being equal to the value of <index>. Secondary entries are used to specify countries which are contained within multi-country World Numbering Plan Areas (e.g, "1" for North America and Caribbean).

The <nextcountry> parameter is used by the **lineTranslateCaps** function when reading in the [Countries] section to build the table of countries in the LINETRANSLATECAPS structure. Country codes are non-contiguous, and this parameter allows the **lineTranslateCaps** function to read in the entries without either guessing at country codes or having to read in the entire section at once and parsing out the entries itself. The last country in the list shall have its <nextcountry> parameter set to 0.

"<samearea>", "<longdistance>", and "<international>" are the dialable string templates for the three classes of calls which may be direct dialed from this country. A parameter should be left empty ("") if that type of call cannot be direct dialed. The permitted contents of these strings are the same as specified above in the [Cards] section.

Call States and Events

 About Call States and Events

Call States and Events

About Call States and Events

Introduction

A connection is not fully established until both parties are communicating. To reach that point, the establishment of the call goes through several stages, as does the clearing (termination) of the call. A call's events cause it to transition through *call states* as it comes into existence, is used to exchange information, and terminates. This appendix lists the various call states, along with a description of what they mean.

Call-state transitions result from both *solicited* and *unsolicited* events. A solicited event is one caused by the application controlling the call (as when it invokes TAPI operations), while unsolicited events are caused by the switch, the telephone network, or the actions of the remote party. Some operations on line devices, addresses, and calls may first require that the line, address, or call upon which they operate be in certain specific states.

Different call states indicate that connections exist to different parts of the switch. For example, a *dial tone* is a particular state of a switch that means the switch is ready to receive digits.

Whenever a call changes state, the service provider reports the new state in a callback to TAPI.DLL. Due to the asynchronous way in which these reports arrive and are forwarded to TAPI.DLL's client applications, the programming model the application developer should follow is not one that preassumes a rigid call state machine, but one where the application should react to the events reported to the application. In other words, call-state notification tells the application what the call's new state is instead of reporting the transitions between two states.

Some of the call states and events defined by TAPI are exclusive to inbound or outbound call processing, while others occur in both cases. Several of these call states provide additional information that can be used by the application. For example, the *busy* state signifies that a call cannot be completed because a resource between the originator and the destination is unavailable, as when an intermediate switch has reached its capacity and cannot handle an additional call. Information supplied with the *busy* state includes *station busy* or *trunk busy*. Station busy means that the destination's station is busy (the phone is offhook), while trunk busy means that a circuit in the switch or network is busy. The call states defined in Windows Telephony are listed below:

Call State	**Description**
idle	This corresponds to the “null” state: No “activity exists” on the call, which means that no call is currently active. This state occurs at the end of a call, but never at the beginning of a new call.
offering (inbound)	When the switch informs the computer of the arrival of a new incoming call, that call is in the offering state. Note that *offering* is not the same as causing a phone or computer to ring. When a call is offered, the computer is not necessarily instructed to alert the user. ***Example:*** An incoming call on a shared call appearance is offered to all stations that share the appearance, but typically only the station that has the appearance as its primary address is instructed to ring. If that station does not answer after some amount of time, the bridging stations may be instructed to ring as well.
accepted (inbound)	An application has taken responsibility for an incoming call. In ISDN, the *accepted* state is entered when the called party equipment sends a message to the switch indicating that it is willing to present the call to the called person; this has the side effect of alerting the users at both sides of the call: the caller’s and the called party’s. An incoming call can always be

	immediately answered without first being separately accepted.
dial tone (outbound)	Indicates that the switch is ready to receive a dialable number. In most telephony environments, this state is entered when audible dial tone is detected by the line device. Additional information includes:
— *normal dial tone*	The “normal” or everyday dial tone, usually a continuous tone.
— s*pecial dial tone*	A special dial tone is often used to signal certain conditions such as message-waiting. This is usually an interrupted dial tone.
dialing (outbound)	The originator is dialing digits on the call. The dialed digits are collected by the switch.
proceeding (outbound)	The call is proceeding through the network. This occurs after dialing is complete and before the call reaches the dialed party, as indicated by ringback, busy, or answer.
special info (outbound)	The call is receiving a special information signal. A special information signal precedes a prerecorded announcement indicating why a call cannot be completed. Such announcements can be of these types:
— *no circuit*	A no-circuit or emergency announcement.
— *customer irregularity*	This typically means that the dialed number is not correct.
— *reorder*	A reorder or equipment-irregularity announcement.
busy (outbound)	The call is receiving a busy signal. Busy indicates that some resource is not available and the call cannot be normally completed at this time. Additional information consists of:
— *station busy*	The station at the other end is off-hook.
— *trunk busy*	The network is congested. This usually produces a “fast busy” signal.
ringback (outbound)	The station to be called has been reached, and the destination’s switch is generating a ring tone back to the originator. A ringback means that the destination address is being alerted to the call.
connected (inbound and outbound)	Information is being exchanged over the call.
onHold (inbound and outbound)	The call is currently held by the switch. This frees the physical line, which allows another call to use the line.
conferenced (inbound and outbound)	The call is a member of a conference call and is logically in the connected state (to the conference bridge). Every call in the *conferenced* state is linked to the parent conference call, which will be in a state such as *connected*, *onHold*, and so forth).

on hold pending conference (inbound and outbound)	The conference call is currently on hold and waiting for the user to add another party.
on hold pending transfer (inbound and outbound)	The call is on hold in preparation of being transferred.
disconnected (inbound and outbound)	The call has been disconnected by the remote party.
unknown (inbound and outbound)	The call exists, but its state is currently unknown. This may be the result of poor call progress detection by the service provider. A call state message with the callstate set to unknown may also be generated to inform the TAPI DLL about a new call at a time that that the actual call state of the call is not exactly known.

Although under normal circumstances an outbound call is likely to transition to *connected* through a number of intermediate states (such as *dial tone*, *dialing*, *proceeding*, and *ringback*), other paths are possible as well. For example, the *ringback* state may be skipped, such as when a “hot phone” (or other non-dialed phone) transitions directly to *connected*.

An application should always process call-state event notifications. Call-state transitions valid for one switch or configuration may be invalid for another. For example, consider a line from the switch that (using a simple Y-connector) physically terminates both at the computer and at a separate phone set, creating a party line configuration between the computer and the phone set. The computer termination and, therefore, the application using TAPI, may not know of the activities on the line handled by the phone set. That is, the line may be in use without the service provider being aware of it. An application that wants to make an outbound call will succeed in allocating a call appearance from the API, but ends up sharing the active call on the line. In this case, blindly sending a DTMF dial string without first checking for a dial tone may not result in intended (or polite) behavior.

Call State Functions and Messages

LINE_CALLSTATE is the message sent to an application to notify it about changes in a call's state. The function **TSPI_lineGetCallInfo** returns mostly constant information about a call as a data structure of type **LINECALLINFO**. The function **TSPI_lineGetCallStatus** returns complete call status information for the specified call as a data structure of type **LINECALLSTATUS**.

The call information data structure maintained for each call contains an application-specific field that applications can read from and write to. This field is not interpreted by the service provider. Applications can use it to “tag” calls in application-specific ways. Writing to this field so that the change is visible to other applications requires that the application use **TSPI_lineSetAppSpecific**, which sets the application-specific field of a call’s information structure.